French Silk smooth ware

P.O. Box 207, Cannon Falls, MN 55009

INSIDE THE VIC™

By: Don French

Contents

Introduction

This book has been written both as a general purpose VIC 20 and 6502 microprocessor tutorial and as a specific complement to and guide for the use of a set of software tools. The tools, along with this book comprise Develop-20, a product of French Silk. While this book is certainly useful by itself, its value may be magnified considerably when used in conjunction with the other tools. The five software tools are The Assembler, The Editor, The Loader, The Monitor and The Decoder. Each of the tools is described in detail in the second section of this text. Cassette tape is the usual distribution media for the tools. However, a disk version is also available as is an extended feature version. Included in the back of this book is an exchange offer for those who wish to upgrade their toolkit.

This book is designed to satisfy the needs of programmers of several levels of experience, including the very inexperienced. It is written in three sections.

The first section gives a detailed look at the architecture of the 6502 microprocessor. Its addressing modes, register set and instruction set are examined. The nature and structure of data is also explored. The introduction to assembly language is presented in this section.

The second section is directed at providing information on the use of the software tools which are an integral part of this package. It will bring you step by step through the mechanics of creating, modifying, running and debugging your programs.

The third section is very VIC-specific. It provides the information which is necessary to utilize the VIC's built-in programs. It provides memory maps of all of the VIC's BASIC operating system along with the information as to how to use some of the built-in programs. It will tell you how to build custom characters and how to make auto-start cartridges and to program for game paddles, joysticks, bit-image graphics and to make sound effects. The appendixes contain additional information on the 6502 and the VIC along with some useful tables and sample programs.

None of the programs in the Develop-20 Kit are protected, so you may make copies for your own use. Information is provided in Chapter 6 on making backup copies of the software tools. You are encouraged to do this as soon as possible so as to identify any problems with the magnetic media within the thirty day warranty

period. With the exception of The Loader, you do not have our permission to remove the copyright notice from the copies or the original. It is specifically prohibited to make copies for resale or for distribution to friends, relatives, associates or anyone else. We hope you will respect the legal and ethical restrictions which apply to the theft of software, both ours and everyone else's. While this product provides tools which make such unscrupulous activities easier to accomplish, it is not the intent of French Silk that you use these tools for this use. We oppose and wish to very strongly discourage the pirating of software. We would also like to point out that there are severe penalties associated with copyright violation including fines up to $10,000 and imprisonment of up to one year not to mention civil liability. In the interest of promoting the continued development of high-quality and low-cost software the consumer must play his and her roles in helping eliminate this source of revenue loss to the software developer. And, as a final argument, you, the owner of this kit are now a member of the community of software developers. It is in all our interests to help discourage the continued growth of this problem. Thank you.

Many people enquire as to the origin of the name French Silk. There are several roots. The very first machine known to have been invented on planet Earth with the ability to contain a stored and modifiable program to control its operation was invented in 1801 by the French silk weaver, Joseph-Marie Charles Jacquard. It was known as the Jacquard loom and its programs were encoded as a series of punched holes on paper cards, the dominant means of program storage for the next 170 years. The invention enabled the creation of beautiful and intricate patterns in the finest fabric known to man and was the precursor of the modern computer. The final, forcing factor in the selection of the name was the combination of the founder's surname and his descendency from a line of silk weavers. (The coincidental existence of a delicious and rich confection by the same name had really very little to do with it).

The author would like to express thanks to Jim Butterfield and Tom Court for their helpful suggestions and criticisms of the early versions of this text.

The 6502 — A Data Processor

As this kit is focused on the development of software through the creative use of machine language, it will be necessary to first understand the machine before learning its language.

You're probably aware that there is somthing inside the case of the VIC-20 which is known as the 6502 microprocessor. This is the heart of the VIC. It is also the heart of the Apple, the Atari, the PET, KIM, AIM, SYM, OSI and a few other microcomputers. It is a product of MOS Technology, a wholly owned subsidiary of Commodore Business Machines.

The 6502 is an integrated circuit. That is, it is a single chip of silicon which has built into it, sort of like etched onto it, the electronic circuitry which connects thousands of microscopically small electronic components. These components are deposited on the silicon by some marvel of modern technology which is beyond the scope of this text. We won't go into how the electronics are created or how they function electronically. We are interested here in how to use this machine and how it fits into the environment of the VIC-20 personal computer.

The piece of silicon called the 6502 is packaged in a piece of plastic or ceramic material about one inch by two inches. It has 40 little bug-like legs called pins which connect the internal circuitry to the outside world. These pins plug into a circuit board which has other similar appearing chips of silicon, each with its own set of pins and its own internal characteristics, different from the characteristics of the 6502. Each of the chips has its own specific function and together they are combined, through their connecting pins and the circuit etched on the printed circuit board into which they are plugged, to make a microcomputer.

This will become more and more clear as we describe what each of the component chips are for and how they work and how they communicate with one another. The description of the functional characteristics of the 6502 will completely define the processor from the programmer's standpoint. The electrical or electronic characteristics are of no interest to us as we have no need to understand the machine at that level.

The 6502 is a data processor. It is a machine which performs simple operations of data manipulation under the control of a stored program. Both the data and the program are stored in memory devices

which are integrated circuits electrically connected to the 6502. Programs are a special class of data and we will explain programs after we discuss data in a more general sense.

Data is information. It can be the balance of your checking account or the grade your physics teacher gave you or the position of PAC-MAN on your video screen. Information, as it is stored inside the computer's memory devices is coded by a special set of simple rules. Memory devices are composed of thousands of cells, or storage locations, where the data is kept. Each cell is composed of eight switches which can be turned either on or off. When the letter "A" is pressed on the keyboard of your VIC, some electronic circuitry will automatically create a pattern of eight switch settings which is uniquely identified as the pattern for the letter "A". This is the code for "A". Every character has its own code and it is different from all the other character's codes. There are only 256 different unique codes which can be constructed from eight switch settings. There are therefore only 256 possible different characters which can be represented and stored in the memory chips of the VIC. That's sufficient to handle A-Z, 0-9, all the special characters and the graphics characters.

These switches are usually called "bits". Bit is short for binary digit. A digit may have 10 possible values, 0-9. A bit may have two possible values, 0-1. A bit which is turned on may be thought of as having the value of 1 and if it is off, it is a 0. So characters are represented as a string of eight bits with bit values of either 0 or 1. The actual bit string for the letter "A" is 01000001. The coding scheme used is an international standard called ASCII, which stands for American Standard Code for Information Interchange.

The 6502's data link with the memory devices is called the data bus. This is nothing more than a set of eight lines, or electrical connections, between the memory chips and the 6502. When the 6502, under control of a program, wishes to either transfer data to or from a memory device, it sends an electrical signal on the R/W (Read/Write) line, telling the device which direction the data is to go. Since the memory device can store thousands of characters of data, it is necessary for the 6502 to tell it which storage location it wishes to get data from or send it to. It does this through another set of electrical connections called the address bus. Every storage cell within the memory device has a unique address associated with it and

when the signal comes to do a data transfer, the memory device is designed to use the information sent on the address bus to know which cell is being selected. Each data cell holds eight bits of information. This basic unit of data, called a byte, is transferred all at once along the eight parallel electrical connections known collectively as the data bus.

The address bus is very similar to the data bus except it has sixteen parallel electrical connectors. Like the data bus, the address bus information is coded in binary. That is, each of the sixteen lines may have only one of two possible states, a 0 or a 1, presented to the devices as 0 or +5 volts. The buses are connected directly to the pins which go inside the plastic package and connect to the internal microcircuitry on the silicon chips. The sixteen bit address bus allows for 65536 different addressable memory locations where data may be stored.

Finally, there is a control bus which contains lines which help to control the various chips in the VIC. The R/W line mentioned above is one of the control signals. With the exception of the interrupt lines, discussed later, we don't need to know much about the control bus.

Because the 6502 transfers and processes data eight bits at a time, it is known as an eight bit parallel processor. The next chapter will go into more detail on the format of the data as it is stored in the memory devices.

Memory devices come in two basic varieties as of this writing. ROM is read-only memory. When the power is turned off ROM doesn't lose its contents. ROM is a kind of chip which may be "Read" but not written to. Its contents are "burned-in" at the factory. There is a similar kind of memory device called a PROM which stands for Programmable ROM and it may be modified by a special piece of hardware called a PROM programmer. It must be erased by shining an intense ultraviolet light on its top surface for some prescribed length of time.

Both of these differ from the other main kind of memory device which is called RAM. RAM is badly misnamed. It should be called MOM for MOdifyable Memory or RAW for Read And Write. RAM stands for Random Access Memory, which means you can extract data from it in any sequence you want. The same thing is true of all currently available types of memory, including ROM.

Anyway, the differences between RAM and ROM is that a program

can write to RAM and change its contents and when the power is turned off the contents of RAM are lost, whereas ROM cannot be modified by any program under any circumsatances and when the power is turned off, the contents of ROM are kept intact. In the VIC, all of the operating system programs and the BASIC interpreter are in ROM. That is why you can run BASIC programs as soon as you turn the machine on, without having to load anything from tape or disk.

The Data

Bits. Binary digits. Its actually a contradiction in terms. Binary means it can have two possible values. Digit implies ten.

The bit is the most basic unit of information. It is the foundation of all other more complex information formats. It is the only kind of information which may be stored in a "digital" computer. A bit may have the value of either one or zero. A byte is a grouping of eight bits. The 6502 is called an eight-bit microprocessor. This is because it processes and stores data eight bits at a time. It is more convienient for the purpose of understanding the nature of computer data to break the eight bits into two four bit sections, called nybbles.

There are exactly sixteen unique four-bit combinations of ones and zeros such that no two arrangements are alike. There is a common shorthand notation for identifying these sixteen patterns. It goes like this:

```
8421     8421     8421     8421
----     ----     ----     ----
0000=0   0100=4   1000=8   1100=C
0001=1   0101=5   1001=9   1101=D
0010=2   0110=6   1010=A   1110=E
0011=3   0111=7   1011=B   1111=F
```

It's a convienient shorthand system because it is easy to remember. It simply numbers the patterns from 0 to 15, except, in keeping with the idea of using a one character code for each pattern, the numbers 10-15 are called A-F. So, pattern 12 is called "C" and 15 is "F", etc. It's got another advantage too. The shorthand label system has an order which makes it easy to remember. If the left-most bit position may be considered to have the value of 8; and the next position, the value 4; and the next, 2; and the last, one: then the bit patterns may be converted to their labels by adding the bit values of the individual bit positions. For example: 1010 has a 1 in the 8 position, a 0 in the 4 position, a 1 in the 2 position, and a 0 in the 1 position. So its label is "A" because 8 + 2 = 10 and "A" is the code for 10. Likewise, 0110 is labeled "6" because it has no 8, one 4, one 2 and no 1.

The leftmost bit position is called the "most-significant" bit because it has the greatest bit value. It is also called the "high-order" bit. Likewise, the rightmost bit is the "least significant" or "low-order bit".

Note that all the odd numbers have a one in the rightmost bit position. All the even numbers, a zero.

You can now see that the shorthand notation for identifying a four bit data field is very logical and it is not hard to convert back and forth between the bit patterns and the pattern code. This is important because as machine language programmers, we have to end up working a lot with bits. This is because the 6502 and all of the memory devices store and work with all data coded as bits.

The addresses of where data is located within a memory device is likewise represented and transferred on the address bus in bits.

Now, since data is stored and retrieved to and from the memory devices eight bits at a time (called a byte of data), it takes two shorthand codes to describe the bit content of the data. A byte of data which has a bit structure of 1100 0100 would be identified as having the two-character code of "C4". The bit structure of the code for the letter "A" would be "41" because the bit pattern assigned to "A" in the ASCII coding scheme is 0100 0001. It's helpful to sit with a pencil and paper and write out bit patterns and figure out what their codes are. You should also start with the codes 0-F and convert back to bit patterns.

That there are 256 ways to represent eight bits is obvious from the fact that there are sixteen possible first characters of the two character code which identifies an eight-bit pattern and sixteen possible second characters. And 16 times 16 = 256. If we wanted a byte of data to represent a numeric value instead of an ASCII coded character, we can see that it could represent any value between 0 and 255. The 6502 does, in fact, sometimes treat data as if it is numeric instead of character data. Using the same two-character code to identify an 8 bit pattern, we can convert between the numeric value and the two-character code quite easily. For example, to convert from "A6" to its numeric value, we would multiply 10 times 16 ("A" = 10) and add 6 to give 166. Likewise, to convert "4F" to numeric we would multiply 4 times 16 and add 15 ("F" = 15) to give 79. It's easy to see that the maximum value of "FF" = 15 times 16, (240) plus 15 to give 255. Of course, the minimum would be "00" which is (0 times 16) plus 0.

How about going the other way? If we want to create the bit pattern for some number in the range of 0 - 255, we only have to divide the number by 16 to get the first character of the code (remember to translate 10 thru 15 to A thru F). Then the remainder of the division is the second character. Now to get the bit pattern we just look it up in the little bit pattern table. Put the first four-bit pattern together with the second and you have an eight bit pattern which represents the numeric value. Not hard at all. Lets try it with a few numbers. Take the number 169. How many times does 16 go into 169? Right. And how is that represented in our single character code? As the letter "A". The remainder from dividing 169 by 16 is 9. So our character-code representation of 169 is "A9" and if we go to the bit pattern table, we will see that the bit patterns are 1010 1001.

It is a valuable exercise to practice converting bit patterns to character-codes and from character-codes to decimal and from decimal to character-codes and character-codes to bit patterns. You will find a table of all 256 bit patterns and the corresponding character-codes and corresponding decimal values in Appendix B. You may check your success with the table.

The character-code system for identifying the various bit patterns is, as you might know or have figured out, what in computer circles is called hexadecimal. Numbers, when represented in hexadecimal, are usually preceeded by a "$". $41 is the hex representation for the ASCII code for the letter "A". This removes any question as to whether the number is decimal or hexadecimal. From now on we will follow that convention in this text.

There is another convention which is widely used to identify the individual bits in a byte. The bits are numbered 0-7 from right to left. The high-order bit is the 7-bit and the low-order bit is the 0-bit.

We discussed the way to represent the decimal numbers from 0 to 255. The same general technique may be used to represent numbers from 0 to 65535. Instead of two hex characters representing one byte of data, we need four hex characters representing sixteen bits or two bytes. Now the first pair of hex characters may be followed by any of 256 pairs of hex characters ($00 - $FF). So the total number of possible combinations of four hex characters is 256 times 256 or 65536. The address bus is sixteen bits wide. Which is why there are exactly 65536 unique addresses possible. Now, it is frequently useful

to be able to convert decimal addresses into hexadecimal. We saw how to do it with a single byte of two hex characters (8 bits). We divided the number by 16 to get the first character and the remainder was the second character. The process is similar for going from a number larger than 255.

We divide the number by 256 to get the first half of the answer. This will be a number between 1 and 255. The remainder will be the second half of the answer. Both of these decimal numbers can then be converted to their hexadecimal counterparts by the dividing-by-16 technique. These hex digits can be then easily converted to bits (binary) by looking up the table or retrieving it from our biological memory device.

Lets do an example: Say we want to get the binary value (bit configuration) of the decimal number 47892. The first thing we do is see how many times 256 will go into 47892. The answer is 187. The remainder is 20. A line of BASIC code to do this computation would be:

```
HA = INT ( NUM / 256 ): HB = NUM - HA * 256
```

HA is the first half of the answer in decimal. HB is the second half. We still have to take 187 and 20 and break them into their two hex components. We do this by dividing by 16. 187 / 16 = 11 with a remainder of 11. So the first two hex digits are $BB. 20 / 16 = 1 with a remainder of 4. So the complete answer is $BB14. The binary equivalent of $BB14 is 1011 1011 0001 0100. To double check our answer, we go back the other direction and convert $BB14 to decimal. $14 = (1 * 16) + 4 = 20. $BB = (11 * 16) + 11 = 176 + 11 = 187. (187 * 256) + 20 = 47872 + 20 = 47892. And that's the number we started with. This process should be practiced. It is very helpful in solidifying the understanding of hex and binary and their relationships to decimal numbers.

The Processor

The 6502 is a machine. Its moving parts are electrons and the only work it does is with data. It has some internal data storage which is identical in nature to the data storage in the memory devices to which it is attached through its external connectors. The internal storage is known as the machine's registers. The registers are eight bits wide and there are only seven of them. Each register has some special characteristics in the way the 6502 can utilize the data contained in it. The data processed by this machine is stored in the external memory and the registers. Processing consists of manipulating the data in some logical sequence which results in accomplishing some desired goal. The 6502 processor does its processing of data by interpreting and executing "instructions".

Instructions are data. They are stored in the memory devices as eight bit bytes which are precoded (programmed) to make the 6502 do some desired operation. The first byte of each instruction is called the Operation Code. The 6502 has pre-programmed circuitry built in to its microelectonics, etched onto its silicon chip, which can decode operation codes and figure out what it is supposed to do next based on the bit structure of the operation code. This logic comes with the 6502. The bright electronic engineers who designed the 6502 at MOS Technology back in 1975 figured out how to make the 6502 interpret bit structures and to take whatever action each operation code was designed to make it do. They preplanned a set of data manipulation operations which they thought would be useful for a microprocessor to be able to do and then set about designing the machine and the operation codes so that those operations could be interpreted and performed. The way the processor is programmed by you the programmer is for you to place in the memory of the computer a sequence of instructions designed to make the 6502 do some presumably useful task. You choose the instructions carefully from the set of available instructions which the 6502 can perform. You code these instructions in the language which the machine can understand, machine language. Each instruction has a specific bit pattern which is understood by the 6502 to mean perform some operation and use one of the 13 addressing modes.

If you have an assembler you can write the 6502 instrucions in an understandable format which makes some sense to humans when they read it. The assembler will then convert the human understandable

program into a machine undersandable sequence of bit patterns (bytes). The Assembler is such a program for the VIC-20. The next section tells you how to use it. In this section we will explore the 6502'architecture, its registers, its instruction set and its various addressing modes. First, the registers.

The Program Counter

The 16-bit Program Counter Register is actually two 8-bit registers, the PCL and PCH registers. These two registers are always used as a pair. The "PCH" stands for Program Counter High and "PCL" stands for Progran Counter Low. Together, they are used by the 6502 to form a sixteen bit address pointer. The 6502 moves the contents of these two registers to the address bus when it wants to fetch an instruction from some memory chip attached to the 6502.

The Program Counter tells the 6502 where the next instruction to be executed is located in memory. When the computer is turned on, an initialization process occurs automatically. This process includes moving the data contained in addresses $FFFC and $FFFD directly into the PCL and PCH respectively. Addresses such as this which are stored in memory and point to the starting point of some other program are called "vectors". This is how the 6502 finds the address of its first instruction to be executed. So, every 6502 must have the address of the beginning of the first program to be executed prestored at $FFFC, $FFFD. This must obviously be in ROM.

This initialization process occurs at power-on time and whenever the RESET button (available on some expansion chassis but not on the standard VIC) is pushed. After each execution of an instruction by the 6502, the Program Counter is incremented to the next instruction, and so the program flow occurs.

The A-reg

The A-reg may be thought of as the Arithmetic register. It is often called the Accumulator. Like all the registers, it is a one-byte (8-bit) register. It is the place where arithmetic operations occur. Instructions like ADC (Add with Carry) and SBC (Subtract with Carry) cause data to be added to or subtracted from the A register. The A reg may be loaded (new value brought into it) with an instruction such as LDA (Load the A-reg). Its contents may be stored

out to some memory location with an instruction such as STA (Store the A-reg). The address in memory where data is loaded-from and stored-to is specified by further addressing information provided to the 6502 by the program in a manner discussed in the next chapter.

References to "STA" or "LDA" instructions are referring to the assembler language English-like mnemonic which gets translated into a machine language instruction recognizable by the 6502. There is a one for one correspondence between assembly language statements and machine language instructions.

The X and Y registers

There are two other "working" registers called the X and Y registers. These are also known as the index registers. These are all eight bit storage registers in the 6502 chip itself. The X and Y registers are used mostly in addressing functions as explained next chapter. The X and Y registers may both be loaded from memory with instructions such as LDX and LDY. They may be saved (stored) in memory with STX and STY instructions. They may also be incremented, decremented, and compared to data in external memory.

Stack Pointer (SP)

The second 256-byte block of memory ($0100-$01FF) is used by the 6502 in a special way. It is called the Stack. The Stack is a special storage block which is automatically utilized by certain instructions. Its primary reason for existence is to allow subroutine "nesting" and to allow for the smooth handling of interrupts.

Subroutines are program segments which can be executed by many different programs. They are sub-programs, which are jumped-to and returned-from. They save having to write commonly used program segments over and over again. The stack is the 6502's communication mechanism for remembering where a subroutine was "called" from. The JSR (Jump to Subroutine) instruction is explained in detail in Chapter 5. Suffice it here to say that the JSR causes the program to jump to a subroutine in such a way that the subroutine can return to the instruction after the JSR once it is done doing its processing. The 6502 saves the return address on the stack when a JSR occurs. It pulls it off the stack when the RTS (return from Subroutine) occurs. The position of the next available stack location for recording return

addresses is kept in the SP. The SP is initialized by the VIC start-up program to the value of $FF at power-on and RESET time. The high-order byte of the stack address is always $01. This is fixed inside the 6502. The first time a JSR instruction is executed, the address of where to return to is pushed onto the stack at $01FF and $01FE. The SP is then decremented by two so that the new value of the SP is $01FD. The subroutine called by the JSR may then call additional subroutines and the return addresses will be stored below the initial return address. There may be up to 128 levels of subroutines calling other subroutines. Each subroutine must have as its last instruction a RTS (return from subroutine) instruction which causes the 6502 to load the PC with the saved address from the stack and to increment the SP by two. Thus the return to the address from which the subroutine was called.

Interrupts are caused by an electrical signal to the 6502 from the outside world. There are two interrupt lines attached to the pins of the 6502. One of them, the NMI line, will cause the 6502 to be interrupted regardless of what it is doing. This is called the Non-Maskable Interrupt. This facility is provided so that hardware which has critical timing requirements may cause the 6502 to service them immediately. It is also provided as a means of unconditionally breaking into the processing of the machine if it is suspected that somthing has gone awry in a program and there is no other way to sieze control of the machine short of turning it off and turning it back on again.

The other kind of interrupt is a maskable interrupt. A means exists to prevent the interrupt from being serviced. An interrupt may be "masked" (made so it can't be seen by the 6502) by the means of an "interrupt disable" bit in the Processor Status register. Maskable interrupts are those whose electrical connections are to the IRQ pin of the 6502.

Both kinds of interrupts are handled in about the same way by the 6502. The 6502 finishes processing the instruction which was in progress when the interrupt signal was recognized. It then saves the PC on the stack just as if a JSR had been executed. Additionally, it saves the Processor Status Register on the stack and decrements the stack by three. It now loads the PC with the address found in location $FFFA and $FFFB for a NMI interrupt or $FFFE and $FFFF for a non-maskable interrupt. These locations must have been pre-programmed to contain the addresses of the programs which were written to service

the interrupts.

The interrupt processing routines must be exited via a RTI instruction (Return from Interrupt). This acts like the RTS instruction except that the Processor Status Register is reloaded from the stack before the PC is pulled from the stack. In this fashion, the interrupted program may continue where it was interrupted and the status of the machine will be as it was at the time of interruption.

The stack is also used by the PHA and PHP instructions to store the A-reg and the P-reg respectively in the stack. This is used to pass infornmation to the subroutine. More information about these instructions may be found in Chapter 5.

The Processor Status Register

The Processor Status Register is a collection of eight bits, somtimes called flags, which reflect and control the operation of the 6502. The bit assignment of the P-reg is:

```
P-reg bit position     76543210
Status-bit label       NV BDIZC
```

The Negative Bit

"N" is the Negative-bit. It is turned on by the processor upon the execution of certain instructions. It reflects whether the result of an addition or subtraction is negative or not. A 1 value in the Negative bit indicates a negative result. This bit is set by load instructions and compare instructions too. See the chapter on the 6502 instruction set for a complete explanation of each instruction an how each affects the various status bits.

The Overflow Bit

The "V" bit is the overflow bit. It reflects whether or not "two's complement" overflow has resulted from a SBC (Subtract with Carry) instruction. It is also set by the BIT instruction. See the descriptions of those instructions for a more complete explanation.

The Break Bit

The "B" bit is set by the processor when a BRK instruction is executed. The BRK instruction causes an interrupt to occur. It is a software interrupt, used mostly in debugging machine language programs. The BRK interrupt is processed almost like a maskable interrupt. The same vector is used by the 6502 to find the address of the interrupt proccessing routine. The only way the processing routine can know if the interrupt was a software or hardware interrupt is by examining the "B" bit in the Processor Status Register which has been stored on the stack. The BRK is not maskable, but it uses the maskable interrupt vector. Also, the return address stored on the stack is the address of the BRK instruction plus two.

The Decimal Mode Bit

The "D" flag in the P-reg is a cue to the processor to do all ADC and SBC instructions in Decimal mode. In decimal mode, the data being added or subtracted will be assumed to be composed of two decimal digits per byte. Each digit is coded as a four-bit pattern having the range of values $0 - $9. The range of values which can be contained in a byte is 0-99 decimal. The name of this data type is Binary Coded Decimal (BCD).

If the Decimal mode is clear, the ADC and SBC instructions will treat the data being added and subtracted as eight-bit binary values in the range 0-255.

The "D" bit may be set and cleared by the program with SED and CLD instructions.

The Interrupt Disable bit

The "I" flag is the interrupt-disable flag. It may be set with the SEI instruction and cleared with the CLI instruction. When set, only non-maskable and software interrupts may occur. When Clear, all interupts are enabled.

The Zero bit

The "Z" flag is set like the "N" flag, by arithmetic and load and compare instructions. If the result of these operations gives a zero result, the Z-flag is set. Otherwise it is cleared.

The Carry bit

The "C" flag is the Carry bit. It is set by addition, subtraction, shift and compare instructions as well as the specific SEC and CLC instructions.

The N, V, Z and C bits may all be tested by conditional branch instructions. A full explanation of this facility is provided in the following chapters.

Processing

The instructions within the program which control the processor cause it to either: 1) load data from memory into one of its internal registers or 2) to move data from one register to another or 3) to store data from one of the registers into memory or 4) to modify the data in one of the registers by some arithmetic operation or 5) to cause a change in the flow of the program.

The 6502, as it decodes each instruction after having fetched it from memory, discovers three things about the instruction: 1) the number of bytes the instruction takes in memory; 2) the addressing mode of the instruction; 3) the operation to be performed. The operation code takes only one byte for every instruction but some instructions need to supply the processor additional information beyond the op-code. This is either address information or data. If, for example, an instruction's purpose is to direct the processor to store the data contained in its A register into some location in memory, it needs to provide the 6502 the information as to where to store it. This could take one or two additional bytes depending on the addressing mode.

Each operation code has coded into it the information as to what addressing mode should be used to accomplish the desired operation. The 6502 has thirteen addressing modes.

ABSOLUTE MODE - Absolute adressing requires a three byte instruction. The first is the op-code and the next two are the two bytes of address information.

You recall that it takes two bytes (sixteen bits) to specify an address in the 6502 address space. This is because the address bus is sixteen bits wide. The first byte of the two byte address is called the high-order byte or the most-significant byte of the address. The second byte is the low-order or least significant byte. These are sometimes abbreviated the MSD and the LSD. The 6502, when it executes instructions with absolute addressing, simply fetches the next two bytes after the op-code and puts them on the address bus when it does the memory access operation specified by the op-code. The memory devices, which are attached to the address bus and the data bus and the control bus, are signaled by the R/W signal and take the address information off the address bus to select the memory location

to either store data into or read data out of. If the R/W signal signifies Write, it will take the data from the data bus and store it into the memory location specified by the address on the address bus. If it is a Read signal, it will take the data already stored at the specified location and load it onto the data bus where the 6502 will find it and do whatever the op-code indicated should be done with it. In absolute mode, the address is stored after the op-code with the least significant byte immediately following the op-code and the most significant byte following that. Example:

```
        LDA $4521
20C0    AD 21 45
```

This is an example of both an assembly language statement on the first line and the machine language following it. The format is the same as that which appears on the screen when you run The Assembler. The assembly language mnemonic is LDA. It is the assembly language equivalent of the op-code. It means Load the A register. The Assembler, which converts assembly language into machine language, decodes the mnemonic and the following operand and produces the machine language which appears on the second line. The Assembler also produces a cassette or diskette file containing the machine language which The Loader can then read and store in the appropriate memory locations, where finally it can be executed as a program.

For now, lets just look at the two statements as they appear here. The $4521 is called the operand field. It specifies to the 6502 where the data to be loaded into the A register is to be found. The 20C0 is the address where the instruction is located. It was arbitrarily picked for this example. The second field, $AD, is the hex representation for the op-code. Following the op-code is $21, the second byte of the address specfied in the assembly statement above. It is followed by $45, the first byte of the address. This is the order the 6502 expects to find addresses. The 6502 processes this instruction when its Program Counter has the value of $20C0. It fetches the op-code at that address and decodes it and executes it in the following sequence: 1) It determines from the op-code of $AD that this is an instruction to cause the A register to be loaded from a location whose address it will find immediately after the op-code. 2) It fetches the next byte after the op-code, $21, and puts it on the least significant byte of the address bus. 3) It fetches the next

byte, $45, and puts it on the most significant byte of the address bus. 4) It sets the R/W line to R. 5) It waits for the memory device to get the specified data and put it on the data bus. 6) It reads the data from the data bus and puts it in its A register. 6. It increases the PCH,PCL register pair by 3 so that it now points to the next op-code.

This is a complete instruction cycle.

ZERO PAGE MODE - If you take the sixteen bit address bus and split it in half, the first eight bits could thought of as a "page number" and the second eight bits could then represent the address (from 0 - 255) within that page. This would mean that there are 256 possible pages, each with 256 memory locations. Zero Page would then represent all memory locations from $0000 to $00FF (0 to 255). Page one would immediately follow, containing the addresses $0100 to $01FF (256 to 511).

These are two pages which have special significance for the 6502. Page zero addresses may be specified with certain machine language instructions which are specifically coded as Zero Page addressing mode. Page one is designated the stack page as explained in the previous chapter. More about that later. The designers of the 6502 decided it would be good to have an addressing mode which allowed the 6502 to execute instructions faster and would consume less memory. The Zero Page addressing mode was part of the solution. The addressing mode is specified as a part of the op-code. In decoding the op-code, the 6502, upon determining that the addressing mode is ZP, then knows that the address to be accessed is in zero page. It therefore has only to load one more byte of addressing data, the low-order or least-significant portion of the address. The high-order half of the address will be forced to $00 by the 6502. Therefore, Zero Page instructions are only two bytes long; the op-code and the address within zero page where the data is to be stored or found. Example:

```
STA ←$7C
20C0 85 7C
```

The left-arrow is the code to the assembler that this is a Zero Page instruction. The address specified by the operand is $007C. This is a Store the A-reg instruction. Op-code $85 causes the contents of the A-reg to be stored into the specified zero page memory location ($007C in this example).

IMMEDIATE MODE - Some instructions may direct the processor to find the needed data immediately after the op-code rather than having to go to some specified address to find it. These are two-byte instructions, one for the op-code, one for the data. They execute even faster than the zero page instructions because the 6502 needn't put anything on the address bus or wait for another fetch cycle to complete before it gets the data it needs. Example:

```
SBC #25
20C0 E9 19
```

Note here that the operand field has a "#" preceeding the data value. This is the code to The Assembler that this is an Immediate Mode instruction. Note also that no "$" preceeds the value 25. The Assembler recognizes four data types, decimal, hexadecimal, address expression and ASCII character. Hex numbers are indicated by a leading "$", ASCII characters by a " ' " decimal numbers by a first character of 0-9, and address expressions everything else. In every case, The Assembler will convert the specified data value into binary (or hex if you wish, the shorthand notation for binary) which is all the 6502 can ultimately understand. The second line displays the machine language in the same format as before. The first field is the address in hex where the instruction will reside, followed by the op-code in hex, followed by the data value in hex also. Check for yourself that $19 is the same thing as decimal 25.

The Assembler has an option you may select each time it is run to print the addresses and generated machine language in either decimal or hex. The file which is created for loading by The Loader will always contain hexadecimal.

SBC stands for Subtract with Carry. It is an instruction to Subtract the specified data value from the contents of the A-reg and to store the result back in the A-reg.

One further note: Most instructions may be specified with a variety of addressing modes. The Assembler examines the operand field to determine which addressing mode is being specified. It then generates the proper machine language op-code to indicate both the

operation to be performed and the addressing mode. As an example, "SBC $4FF3" is an absolute mode version of the SBC instruction and the op-code for it is $ED as compared with $E9 as in the above example. Appendix A contains a list of all instructions and their allowable addressing modes.

IMPLIED MODE - The implied mode of addressing is the fastest executing and the shortest instruction length. In implied mode, only the internal registers of the 6502 are addressed. Beyond the op-code, no more information is required, so implied mode only takes one byte. Examples:

```
 TAX
023F AA
 TAY
0240 98
```

TAX causes the contents of the A-reg to be transferred to the X-reg. TYA causes the contents of the Y-reg to be transferred to the A-reg.

A-REG MODE - The A-reg is sometimes called the Accumulator. This is a carry over from more primitive times. In any event, Commodore and MOS Technology choose to refer to the A-reg Mode as the Accumulator Mode. Call it what you like, it is really an implied mode. In A-reg Mode, only the A-reg and the Carry, Negative and Zero bits of the Status register are affected. The A-reg mode instructions are valid only for the "shift" instructions, ASL, ROL, LSR and ROR. For more information on shift instructions, see the next chapter. Examples:

```
 ROR A
021A 6A
 ASL A
021B 0A
```

RELATIVE MODE - There are three classes of instructions which cause the flow of the program to change. The JMP and JSR instructions are explained in the following chapter. All three types of instructions accomplish program flow changes in the same general way. They cause the Program Counter register to be modified. The PC is the register pair which points to where the next instruction is to be found in

memory. It is automatically incremented by the instruction length each time an instruction is executed. The instructions which modify the PC cause the program to "take a branch". That is, the next instruction to be executed will not be the one immediately following. It will be found at an address which is determined by the addressing data suppied by the branching instruction.

The adddressing information supplied by the "relative mode" instructions is a single byte of data which follows the op-code and which is added to the value of the PC to determine the new PC value. The branch is to an address which is the specified number of bytes away from the branch instruction itself. The address information is called the relative displacement.

These instructions only modify the PC sometimes. They are called conditional branch instructions. They test the status of a bit in the Processor Status Register and the 6502 decides at the time the instruction is executed whether to Branch (modify the PC) or not, based upon the value of the bit being tested. Example:

```
        CMP #'A        0300 C9 41
        BNE NOTA       0302 D0 03
        JMP PROCESSA   0305 4C 7D 04
NOTA    CMP   #'B      0308 C9 42
        BEQ PROCESSB   030A F0 CC
```

This is a short program segment which first compares the contents of the A-reg with the character "A". The format of the program listing is the same as that which is produced by The Assembler when printer output is employed. Note the Immediate symbol, "#" and the "character" symbol " ' ". The function of the compare instruction is to set the Zero bit in the status register if the compare proves to be a match. Otherwise the Zero bit is cleared.

The instruction after the compare is the conditional branch instruction. It is a Branch Not Equal instruction. If the result of the compare results in the zero flag being set, the branch will not happen and the JMP instruction following the BNE will be executed next as usual. If not, the next instruction to be executed will be the instruction at $0308.

Here you see one of the great advantages of having an assembler which allows you to use labels to identify program

locations. You, the programmer can use meaningful symbols to refer to some address in memory. You write the assembly language program without regard to the actual addresses of each instruction in the program. If you want to cause the program to branch to some instruction somewhere in the program, you put a label in English on the instruction and The Assembler automatically computes the address of the instruction for you. In this case the symbolic label is "NOTA".

Assemblers which permit this capability are called symbolic assemblers. Single-line assemblers such as is incorporated in VICMON are severely limited by their lack of this feature.

The generated machine language is particularly interesting here. The byte following the BNE op-code is a "$03". Relative addresssing means that the value found in the byte after the op-code is the number to be added to the PC to find the address to be Branched to. That is, if the 6502 finds that the status bit being tested by the Branch instruction indicates a Branch should be taken, it then adds the value found in the byte after the op-code to the PC. (It has already incremented the PC by two before testing the status bit). The Assembler automatically computes the difference between the address of the instruction following the BNE instruction and the beginning of the instruction NOTA. In this case the amount of adjustment is three bytes. NOTA occurs three bytes past the address of the JMP instruction. It is possible to specify a conditional branch forward by as much as 127 bytes or backwards as much as 128 bytes. When backward branches are specified, the displacement value in the byte after the op-code must contain a negative number.

We mentioned earlier that a single byte may represent the decimal range of 0-255. For special situations such as the relative branch instructions it is convienient to allow the 256 possible values of the eight bits to represent a range of numbers from -128 to +127. To accomodate this need, a system was devised to indicate negative numbers. It is called the two's complement system. It seems a little strange at first, but with a little practice it, too, can be mastered.

Negative one is represented as $FF. Negative two is represented as $FE (or 254 in regular decimal). You can convert a number to its negative by subtracting it from 256 then converting the result to hex. For example, the hex representation of -6 is 256-6=250=$FA. This system is popular with computer designers because it makes arithmetic easy. Note that 256 in hex is $0100. Subtracting

six from $0100 gives $FA. But $0100 takes two bytes. The maximum value of one byte is $FF. If we add one to $FF we get $00 with a carry of one. And with $FF representing -1 it is very nice that when we add 1 to -1 we get 0. Likewise when we add 1 to minus 2 ($FE) we get $FF (-1). Another advantage of this system is that all the negative numbers have the high-order bit on (leftmost bit value = 1). The positive numbers are $00 (0000 0000) to $7F (0111 1111) and the negative numbers are $80 (1000 0000) to $FF (1111 1111). rIt is not coincidental that the N-flag (Negative bit of the Status Register) is set every time any arithmetic operation results in a value with the high-order bit on.

The Assembler will automatically convert relative branch displacements to the proper value for both forward and backward branches. And The Assembler will also allow the expression of negative numbers in the more familiar format of decimal numbers (-34, -122, etc.) and do the conversion to the two's complement value for you. In debugging, however, it sometimes comes in handy to be able to do the conversion yourself and is worth knowing how to do.

INDEXED MODES - The X register and the Y register are sometimes called the index registers. This is because they are used as indexes to data. That is, the relative position of data in a string may be addressed by specifying a base or starting address of the string plus some position index to indicate which data element in the string is being addressed. The X and Y registers can be used with certain instructions to be the position index. The instruction specifies a base address and an index register. The 6502 adds the value of the index register to the base address to get the effective address of the data to be accessed. This is a very useful capability. The following program segment illustrates the use of indexed addressing to move a string of data from one place in memory to another:

```
STRING1 EQU $0400         0400
STRING2 EQU $0480         0480
        EQU $2000         2000
        LDX #10           2000 A2 0A
LOOP    LDA STRING1,X     2002 BD 00 04
        STA STRING2,X     2005 9D 80 04
        DEX               2008 CA
        BPL LOOP          2009 10 F7
```

Several new things are presented in this program segment. The first two statements cause The Assembler to equate a symbolic label with a specific address in memory. For every subsequent reference to the label being EQUated (STRING1 in this example) The Assembler will know that the address being referred to is the one in the operand field of the EQU statement ($0400). Note the generated machine language which follows the statement; LOOP LDA STRING1,X. The address $0400 is automatically generated by The Assembler (in the required low-order-byte-first format).

The third EQUate tells The Assembler what value to assign to the Location Counter. The Location Counter is The Assembler's equivalent of the 6502's Program Counter. The difference is that the Program Counter is an actual hardware register contained on the 6502 chip. The Location Counter is The Assembler's symbolic equivalent of the PC. The address printed on the machine language line after each assembly instruction is the value of the Location Counter for each instruction. It shows us where the generated machine language will be in memory when The Loader finally loads the program into memory. Every EQUate statement sets the Location Counter to the address expressed in the operand field of the statement.

The LDX instruction loads the X-reg with the value 10. The next two instructions illustrate the indexed addressing mode. The first instruction loads the A-reg with a byte of data found at address $400A. The address specified in the LDA instruction is $0400. The contents of the X-reg are added to the specified address by the 6502 before putting the address on the address bus. Since the X-reg has just been loaded with the value 10, the address where the data will come from to be loaded into the A-reg is $0400 + $000A or $040A.

The next instruction turns right around and stores that same data back in memory at the address $048A. So we now have three copies of the same byte of data. One copy in $040A, one in $048A and one in the A-reg. The next instruction, DEX, causes the X-reg to be DEcreased or decremented by 1. Since it had the value of 10 coming into this instruction, after the instruction is executed, it will have the value of 9. The next instruction, BPL LOOP, will conditionally branch to the instruction which has the label LOOP. Note this is a backward branch of 9 bytes. The generated machine language value of how far to branch is $F7. Remember about negative branch displacements? 256 - 9 = 247. 247 / 16 = 15 with a remainder of 7.

Hence the hex value of -9 = $F7. Convieniently, The Assembler made that calculation for us.

The BPL instruction tests the Negative bit of the Status Register. BPL stands for Branch if PLus. The Negative bit is affected by every execution of a DEX instruction (and many other instructions as well). If the X-reg was zero before the execution of the DEX, the result of the DEX would be a negative number in the X-reg. The Negative bit would be set to one. The BPL tests the negative bit. If it is not a one (not negative) the X-reg must still be positive or zero after having been decremented by the DEX.

The branch will be taken back up to LOOP. The A-reg will now be loaded from address $0409 and stored into address $0489. The X-reg will be decremented again and tested to see if it went negative yet. Once again, it is positive (it has the value 8 now). The branch will be taken back up to LOOP, the same process will occur once again, this time moving a byte from $0408 to $0488. Once again, the X-reg will be decremented and tested and found not-negative. The loop will be executed a total of 11 times, with the X-reg varying from 10 to 0, the address of data being loaded into the A-reg varying from $040A to $0400, the address of where data is moved to varying from $048A to $0480.

This is the process by which a wide variety of repetitive operations are performed upon data with the 6502. This is a very standard loop. If it still seems mysterious to you read it over again and when we get to the section on actually using The Assembler, The Editor, The Loader, The Monitor, and The Decoder, we will create an actual program which does just this process. We will execute the program one step at a time with The Monitor, watching how everything works and seeing the registers and the memory locations changing as we go through the program.

There are four modes of addressing which are called indexed. Both the X-reg and the Y-reg may be used in indexed instructions and both may be used in combination with absoulte and Zero Page modes. Like the absolute and ZP addressing modes, the absolute,X and ZP,X instructions take three and two bytes respectively. The Y-reg and the X-reg function identically in their respective modes. The four indexed modes then are: abs,X ; ZP,X ; abs,Y ; ZP,Y .

INDIRECT MODE - There is only one instruction which uses the simple

indirect mode of addressing. This is the JMP (addr) instruction. The parenthesis around the absolute address signifies indirect. What happens with the indirect jump is the following: 1) The 6502 gets an address from the two bytes immediately following the op-code. Rather than load the PC with this value directly, it fetches an address from the specified memory location and the memory location immediately following it. This is the address which it loads into the PC. Thus a change in the program flow is caused.

To state this another way, for the JMP indirect to work as desired, there must be an address pre-stored somewhere in memory. The JMP instruction must tell the 6502 where that address is located in memory. The 6502 will then load its PC with the address stored therein. Such a prestored address is called a vector. BASIC has several vectors saved in the first few pages of memory which point to various processing programs.

Vectors are convienient ways of allowing the flexible design of operating systems such that new versions and updates to the operating system can be compatible with the old versions. Programs which need to use the various routines pointed to by vectors will not need to be changed because of a different location of the routine in the new version. The vector will be the only thing which will have to be modified to allow compatibility.

(INDIRECT),Y - This mode is somewhat similar to the indirect mode discussed above. The above instruction was applicable only to the JMP instruction. This mode is applicable to various data access and manipulation instructions. These instructions are two byte instructions. The second byte of the instruction specifies an address in zero page. Like with the previous mode, there must be an address stored at the specified location. The big difference here is that the Y-reg is added to the address found in zero page to give the 6502 the eventual address of where the data should come from or go to. The (Indir),Y mode is actually used exactly like the addr,Y mode. It is useful for doing loops. The only difference is that the base address of the loop is stored in Zero Page rather than specified directly by the instruction. The instruction then specifies the Zero Page location of the base address. This is a very handy way to program a subroutine which is used at different times and called from different places in the mainline program to do the same general task but with differing sets of data. The base address of the data to be

manipulated by the subroutine must be appropriately set up in Zero Page each time just before the subroutine is called. The subroutine itself never has to change anything. It uses the zero page vector to get the data it's been called to use. Example:

```
SUBRA  LDY ←$40
LOOPA  LDA ($7C),Y
       STA ($80),Y
       DEY
       BPL LOOPA
       RTS
```

Here, the subroutine, SUBRA, is designed to do a general purpose move of data from some location in memory to some other location. The number of bytes to be moved is found in location $0040. The location of where to move the data from is found in $007C and $007D. The address of where to move it to is found in locations $0080 and $0081. The calling program must set those locations up with the appropriate values for the subroutine to function as desired. Note that no left arrow is required for the (indexed),Y mode even though the specified address is always in Zero Page.

(INDIRECT,X) - This is the last and probably the least useful addressing mode provided with the 6502. You may have guessed that it is similar to the previous mode. It would be a lot more useful if it were identical except for the register used. Unfortunately, it isn't. You should notice that the X is inside the parenthesis, whereas in the previous mode, the Y is outside the parenthesis. This is the important distinction. The parenthesis indicate the "indirection". In the previous example, the Y-reg was added to the address found in Zero Page, and the result of that addition provided the address of the desired data. Here, the X-reg is added to the specified address to find where in Zero Page the vector resides. The X-reg is therefore an index to a table of vectors stored in Zero Page. This mode is not useful, therefore, in the same way that the other one is. It might find some use in unusual situations where there is a need to have a list of addresses of data bytes and a routine is needed to process the various data bytes. Such a routine would step through the list, an address (two bytes) at a time, using the X-reg to bump thru the list and to point to the appropriate address at which the data will

ultimately be found. Perhaps you can find some better use of this mode.

These last four chapters have been fairly packed with information on the workings of the 6502. It will be necessary to write some programs and to examine the programs of others to get a comfortable feeling about the use of the various instructions and addressing modes. The next chapter will present every available instruction on the 6502. After completing it, you will be able to start to take control of your computer by writing powerful machine language programs.

6502 - Instruction Set

There are 56 separate instructions which the 6502 has been designed to execute. There are also two "pseudo-op instructions" which are not a part of the 6502's instruction set but which The Assembler understands.

There are 13 addressing modes. Some instructions are limited to a single addressing mode, others are capable of utilizing up to eight. Most of the instructions are quite simple functionally, and require only a sentence or two to describe their characteristics. We will cover these first.

Register-only instructions

TAX - Transfer the contents of the A-reg to the X-reg. Only the receiving register is modified. The Zero bit of the Status Register is set (made to have the value 1) if the value transferred is zero, otherwise it is cleared (made to have the value 0). The Negative bit of the Processor Status Register is set if the high-order bit of the value transferred is on (value 1), else it is cleared.

TAY - Transfer A-reg to Y-reg. The A-reg is stored into the Y-reg. The same notes apply as with TAX.

TYA - Transfer Y-reg to A-reg. The Y-reg is stored into the A-reg. The same notes apply as with TAX.

TXA - Transfer X-reg to A-reg. The X-reg is stored into the A-reg. The same notes apply as with TAX.

TXS - Transfer X-reg to Stack Pointer. The X-reg is stored into the Stack Pointer. No status bits are affected. This is the only way of initializing the Stack Pointer.

TSX - Transfer Stack Pointer to X-reg. The Stack Pointer is stored into the X-reg. The same notes apply as with TAX.

SEC - SEt the Carry bit. The Carry bit is set to value 1.

CLC - CLear the Carry bit. The Carry bit is cleared.

SED - SEt the Decimal Mode bit. The Decimal Mode bit is set to value 1. The Decimal mode of arithmetic is enabled.

CLD - CLear the Decimal Mode bit. The Decimal mode of operation is disabled. The Decimal Mode bit is cleared.

CLV - CLear the oVerflow bit. The overflow bit in the Processor Status Register is cleared.

CLI - CLear the Interrupt disable bit. The Interrupt disable bit in the Processor Status Register is cleared, allowing interrupts to occur.

SEI - SEt the Interrupt Disable bit. The Interrupt disable bit in the Processor Status register is set to 1, causing all interrupts to be masked (disabled) until the Interrupt disable bit is cleared with a CLI.

DEX - DEcrement the X-reg. The value contained in the X-reg is decreased by 1. If the resulting value in the X-reg is zero, the Zero bit is set in the P. If it is not zero, the zero bit is cleared. If the resulting value in the X-reg has the high-order bit on, the Negative bit in the Processor Status Register is set. Otherwise the Negative bit will be cleared. Note that register values of 0-127 will have a clear high-order bit and values of 128-255 will have the bit set.

DEY - DEcrement the Y-reg. The value of the Y-reg is decreased by 1. The Zero and Negative bits are affected as with the DEX instruction.

INX - INcrement the X-reg. The value of the X-reg is increased by 1. The Zero and Negative bits are affected as with the DEX instruction.

INY - INcrement the Y-reg. The value of the Y-reg is increased by 1. The Zero an Negative bits are affected as with the DEX instruction.

Memory accessing instructions

INC - INCrement memory. The value of a byte located in memory is increased by 1. The Zero and Negative bits are affected as with the DEX instruction. Only ZP; ZP,X; ABS; ABS,X modes are valid.

DEC - DECrement memory. The value of a byte located in memory is decreased by 1. The same comments as apply for INC.

LDA - LoaD the A-reg. The contents of a memory location is transferred to the A-reg. The Zero and Negative bits are affected as with the TAX instruction. Valid addressing modes are: Immediate; ZP; ZP,X; Absolute; Absolute,X; Absolute,Y; (Indirect,X); (Indirect,Y)

LDX - LoaD the X-reg. The contents of a memory location is transfered to the X-reg. The Zero and Negative bits are affected as with the TAX instruction. Abs, ZP and ZP,Y addressing modes are valid.

LDY - LoaD the Y-reg. The contents of a memory location is transferred to the Y-reg. The Zero and Negative bits are affected as with the TAX instruction. Abs, ZP and ZP,X addressing modes are valid.

STA - STore the A-reg. The contents of the A-reg is transferred to a memory location. No registers or status bits are affected. Valid addressing modes are: ZP; ZP,X; Absolute; Absolute,X; Absolute,Y; (Indirect,X); (Indirect,Y)

STX - STore the X-reg. The contents of the X-reg are transferred to a memory location. No status bits or registers are affected. Abs, ZP and ZP,Y addressing modes are valid.

STY - STore the Y-reg. The contents of the Y-reg are transferred to a memory location. No status bits or registers are affected. Abs, ZP and ZP,X addressing modes are valid.

Conditional Branch Instructions

BCC - Branch Carry Clear. The program counter will be modified by the amount of the specified displacement if and only if the Carry bit is clear (0). Forward branches may occur up to 128 bytes from the address of the first byte following the branch instruction. Backward branches may occur up to 127 bytes from the same address. No other registers other than the PC are affected. Only Relative addressing mode is valid.

BCS - Branch Carry Set. The program counter will be modified by the amount of the specified displacement if and only if the Carry bit is set (1). Same comments as for the BCC instruction.

BEQ - Branch EQual. The program counter will be modified by the amount of the specified displacement if and only if the Zero bit is set (1). Same comments as for the BCC instruction.

BNE - Branch Not Equal. The program counter will be modified by the amount of the specified displacement if and only if the Zero bit is clear (0). Same comments as for the BCC instruction.

BMI - Branch MInus. The program counter will be modified by the amount of the specified displacement if and only if the Minus bit is set (1). Same comments as for the BCC instruction.

BPL - Branch PLus. The program counter will be modified by the amount of the specified displacement if and only if the Minus bit is clear (0). Same comments as for the BCC instruction.

BVC - Branch oVerflow Clear. The program counter will be modified by the amount of the specified displacement if and only if the Overflow bit is clear (0). Same comments as for the BCC instruction.

BVS - Branch oVerflow Set. The program counter will be modified by the amount of the specified displacement if and only if the Overflow bit is set (1). Same comments as for the BCC instruction.

Jump instructions

JMP - JuMP. The PC is loaded with the specified address, causing a change in the flow of the program. Both Absolute and Indirect addressing modes are valid.

JSR - Jump to SubRoutine. The PC is first incremented by 2. The new PC is then stored on the stack. The PCH is stored in the stack location addressed by the Stack Pointer. The Stack Pointer is decremented and the PCL is then stored in the new stack location as addressed by the SP. The SP is decremented a second time and the address specified by the JSR instruction is loaded into the PC, causing a jump to the specified subroutine location. Note that the address stored on the stack is not what might be expected. The address is of the third byte of the JSR instruction, not the address of the next sequential instruction after the JSR. The RTS instruction, which causes a return from the subroutine, compensates for this anomaly. Generally, the casual programmer needn't worry about the mechanics of stack operations as long as she always has a RTS for every JSR. However, advanced machine language programmers are fond of direct stack manipulation techniques, especially for passing arguments to subroutines.

RTS - ReTurn from Subroutine. The Stack Pointer is first incremented. The PCL is loaded from the stack address pointed to by the Stack Pointer. The SP is then incremented again and the PCH is loaded from the stack. The PC is incremented to compensate for the JSR operation of putting the address of the third byte of the JSR on the stack instead of the address of the next instruction. Now the PC has the address of the instruction immediately following the most recent JSR instruction. The program flow is thus returned to the mainline program from the subroutine.

RTI - ReTurn from Interrupt. This instruction reverses the process which occurs when an interrupt occurs. The Processor Status Register is retrieved from the stack where the interrupt caused it to be stored. The PCH and PCL are then reloaded from the stack where they too were stored as a part of the 6502's interrupt processing. The return to the point of interruption is thus complete, with the Processor Status Register having the same value it had at the time of

interruption.

BRK - BReaK. Interrupt processing is caused to occur. The address of the next byte following the BRK instruction is saved on the stack. The Break bit os turned on in the Processor Status Register which is then saved on the stack. The PC is loaded with the address found at memory location $FFFE and $FFFF.

Stack Push & Pull Instructions

PHA - PusH the A-reg. The A-reg is stored on the stack at the address pointed to by the SP. The SP is then decremented. All PHA's should generally be matched with a following PLA.

PLA - PulL the A-reg. The SP is incremented, then the A-reg is loaded from the stack location pointed to by the new value of the SP.

PHP - PusH the P-reg. The Processor Status register is pushed onto the stack in the same fashion the A-reg is with the PHA.

PLP - PulL the P-reg. The Processor Status register is pulled off the stack in the same fashion the A-reg is with the PLA instruction.

Pseudo-op Instructions

BYT This is not really an instruction in the instruction set of the 6502. It is an instruction which The Assembler recognizes and interprets to mean generate machine language data. The operand field of the BYT instruction can express several types of data which The Assembler will understand.

If the first character of the BYT is a " $ ", the following characters must be hex characters, i.e. 0-9, A-F. The Assembler will handle a string of hex characters up to 80 characters in length. It will generate a data string with two nybbles (a half byte - 4 bits) per byte, inserting a $0 in the high-order nybble of the high-order byte if there are an odd number of characters specified.

If the first character is a " ' ", The Assembler will create a data string with as many bytes as there are characters following the '.

The values of the generated bytes will be the ASCII values of the corresponding characters.

If the first character of the operand is a number in the range 0-9, The Assembler will interpret the operand as a decimal number and convert it to its binary equivalent.

If the first character is none of the above, the operand will be assumed to be an address expression which represents a two-byte address. It will compute the address associated with the expression and generate the standard low-order first format two-byte address. Address expressions are covered fully in the chapter on writing assembly language programs.

EQU - EQUate. This pseudo-op does not generate any data which gets stored into memory by The Loader. It is an instruction to The Assembler to set the Location Counter and to cause the label in the label field to be assigned to the address which is expressed in the operand field.

Shift Instructions

ASL - Arithmetic Shift Left. The contents of the A-reg or of a memory location are shifted one bit position to the left. The low-order bit position is forced to value zero. The high-order bit is shifted into the Carry bit. The Negative bit is set if the bit shifted into the high-order bit position is a 1. It is cleared if it is a zero. The Zero bit is set if the resulting value of the shifted byte is zero, cleared if it is not.

As an example, if the A-reg has the value $CC (1100 1100), after the "ASL A" instruction has been executed, it will have the value $98 (1001 1000) and the carry bit will be set. If the A-reg has the value of $5F (0101 1111) the "ASL A" will cause it to become $BE (1011 1110) and the carry bit will be clear. Note that each left shift causes the A-reg to double in value as long as the high order bit is not a one before the shift. Shifting left is a convienient way of multiplying a value by two. Using ASL in combination with ROL, a multiple precision shift may be effected. See the description of the ROL instruction. Valid addressing modes are: Accumulator; ZP ; ZP,X ; ABS ; ABS,X.

ROL - ROtate Left. The A-reg or a byte in memory may be shifted one bit to the left. The high-order bit gets shifted into the Carry bit. The low-order bit recieves the previous contents of the Carry bit. The same addressing modes apply as for the ASL instruction.

A multiple precision bit shift is one where a string of bytes is treated like one long bit pattern and the shift causes the bits which come out of the high-order positions of one byte get shifted into the low-order positon of the next byte in the sequence. For example:

```
        LDY #3      SHIFT 4 BITS
BITSH   LDX #4      THRU 5 BYTES
        ASL STR,X   RIGHT-MOST BYTE
        DEX
ROLIT   ROL STR,X
        DEX
        BPL ROLIT   ALL BYTES ?
        DEY
        BPL BITSH   ALL BITS ?
```

This routine would cause the string of five bytes at STR to be left shifted four bits. Each execution of the ROL shifts the contents of the Carry bit into the low order bit of the byte being shifted. The Carry bit will contain the bit which was shifted out of the high-order bit position of the previous byte. The ASL is used as the first shift instruction to force zero bits into the low order positions of the low-order bytes.

LSR - Logical Shift Right. The contents of the A-reg or memory location specified is shifted one bit position to the right. The low-order bit gets shifted into the Carry bit. The high-order bit position is forced to zero. The Zero bit is set based upon the resulting value of the shifted byte. The Negative bit is forced to zero. The same addressing modes apply as for the ASL instruction.

ROR - ROtate Right. All bits in the rotated byte are shifted one bit position to the right. The Carry bit is shifted into the high-order bit position and the low-order bit position is shifted into the Carry bit. The same addressing modes apply as for the ASL instruction. The LSR and ROR instructions are the same as the ASL and ROL instructions except they shift bits in the opposite directions. Note that a one-bit shift right results in an effective division by two.

Boolean arithmetic instructions

AND - Logical AND. The A-reg is logical ANDed with the specified data byte. The boolean AND operation is performed between corresponding bits of the two bytes. Each bit position of the pair of bytes is operated on individually, the result of the operation replacing the corresponding bit in the A-reg. The rules of the AND operation are: if the two bits being ANDed are value 1, the result is a 1; if either bit is a 0 the result is a 0. That is, [0 AND 1] = 0 ; [0 AND 0] = 0 ; [1 AND 1] = 1 ; [1 AND 0] = 0. Example:

```
     11001010  Memory
AND  10101100  A-reg
     --------
     10001000  new A-reg
```

Only the bit positions which had a 1 in both bytes ended up with a 1 in the result. The AND instruction is frequently used to selectively clear individual bits while maintaining the status of the other bits in the byte. This is done by creating a "mask-byte" which has a 0 in every bit position which needs to be cleared (set to 0), and a 1 in all the other bit positions. The mask byte may be in either the A-reg or the specified memory location but the result of the operation always replaces the A-reg.

This process works because a 0 ANDed with either a 0 or a 1 gives a 0 result while a 1 ANDed with a 0 gives a 0 and ANDed with a 1 gives a 1.

ORA - Logical OR. The A-reg and the specified memory location are logical ORed together, the result replacing the A-reg. The boolean OR operation, like the AND operation, is a bit by bit operation. Each bit of the A-reg is ORed with the corresponding bit of the byte in memory by the following rules: If either bit is a 1, the result is a 1. If

both bits are 0, the result is 0. That is, [1 OR 1] = 1 ; [1 OR 0] = 1 ; [0 OR 1] = 1 ; 0 OR 0 = 0. Example:

```
          11001010  Memory
     ORA  10101100  A-reg
          --------
          11101110  A-reg
```

The ORA is frequently used to selectively turn bits on (set to 1). Like with the AND, a mask must be created which indicates the desired bits to set and the bits to be unaffected. The OR mask must have a bit on in the bit positions to be set and off in the positions which need to be maintained. This is opposite of the AND mask. There, 1-bits maintained the status quo. Here, 0-bits have that responsibility. A 0 in the mask byte when ORed with a 1 gives a 1 and when ORed with a 0 give a 0. And a 1 in the mask byte always results in a 1 result. [1 OR 1] = 1; [1 or 0] = 1. Valid addressing modes are the same as for the AND instruction.

EOR - Exclusive OR. The contents of the specified memory byte are EORed with the contents of the A-reg, replacing the A-reg with the result. Like the AND and OR instructions, this is a bit oriented instruction. The rules of Exclusive-Oring are: The result will have a 1 in any bit position for which only one of the two bytes being EORed have a one. All other bit positions of the result will have a 0. (i.e. [1 EOR 0] = 1; [1 EOR 1] = 0; [0 EOR 0] = 0; [0 EOR 1] = 1). Example:

```
          11001010  Memory
     EOR  10101100  A-reg
          --------
          01100110   New A-reg
```

The EOR instruction is useful for inverting bits. A 1 in any mask bit position will cause the corresponding bit in the result to have the opposite value as that of the corresponding bit in the object byte. A 1 before EORing will result in a 0 after and vice versa. A zero in the mask byte will cause the corresponding bit in the object byte to go unmolested.

BIT - BIT test. The A-reg is logically ANDed with the contents of the specified memory location. The Zero flag is set if the result of the operation gives a zero result. The Negative bit is set if the high order bit of the memory location is set. The Overflow bit is set if the second-highest-order bit (the 6-bit) of the memory location is set. The Negative and Overflow bits are cleared if the 7-bit and 6-bit of the memory location are clear. The A-reg is not affected by the executioon of this instruction.

This instruction is very useful for testing individual bits of bytes in memory. The A-reg is loaded with a mask which has ones in the bit positions to be tested and zeros in the rest. If any of the memory byte's bits in the tested positions are on the result of the ANDing operation will be non-zero. Note that the mask may be either in memory or in the A-reg for the test to work. The Negative and Overflow bits are set based only upon the bit configuration of the memory byte however. Example:

```
MASK    BYT $06
        LDA MASK
        BIT VAL1
        BMI BIT7
        BVS BIT6
        BNE BIT1OR2
OK      EQU @
```

The value of the byte assigned to label MASK is $06. The 1-bit and the 2-bit are on and all others are off. The BIT instruction ANDs the contents of the A-reg, $06, with the byte at VAL1 and the result will be non-zero only if either the 1-bit or the 2-bit of VAL1 is a one. If VAL1 has its high-order bit on, the program will branch to BIT7. If the 6-bit is on in VAL1 the program will branch to BIT6. If either bit-2 or bit-1 are on, the branch to BIT1OR2 will be taken.

The " @ " is used in the last statement of the program. It has a special significance to The Assembler. Used in the operand field of an instruction it is like a symbolic label except it references the current value of the Location Counter. Here, it performs the function of assigning the value of the Location Counter to the label "OK".

Arithmetic instructions

ADC - ADd with Carry. Addition is performed with the A-reg, the specified memory location and the Carry bit. (A-reg) = (A-reg) + (addr) + (carry). The (...) is used here to mean "the contents of". The result replaces the A-reg. The mode of arithmetic is determined by the status of the Decimal mode bit at the time the instruction is executed. If set, the mode of addition is the Decimal mode. If clear, the mode is binary. Valid addressing modes are thae same as for the LDA instruction.

Binary addition is quite simple. It is just like decimal addition except the highest number you have to worry about is 1. The rules of addition are:

```
0 + 0 = 0
0 + 1 = 1
1 + 1 = 10   (2 in decimal)
```

The third example is the only one which is different from decimal addition. When we add a pair of binary numbers with more than one bit apiece, we proceed from right to left just like decimal addition. We add the two bits together and if the result is greater than 1 we have to carry 1. So, 11 + 11 = 110. Doing this addition one step at a time, taking the rightmost bits, 1 and 1, and adding them by the above rules, we see that the answer is 10, or 0 with a carry of 1. Next we add the next pair of bits plus the carry. 1 + 1 + 1 = (1 + 1) + 1 = 10 + 1 = 11. Thats a 1 with a carry of 1. Note that 1 + 1 + 1 = 3 in decimal and 1 + 1 + 1 = 11 in binary which is the binary equivalent of 3 decimal. The most you have to remember in binary addition of two numbers is 0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 1 = 10 (0 carry 1) ; 1 + 1 + 1 = 11 (1 carry 1). The ADC instruction does this binary addition work for you, so you might not need to know it. On the other hand, you probably will when you go to debug your Galactic Gobbler Game. So you might just as well learn it now.

When in Decimal mode, the 6502 expects the data it is adding to be in "binary coded decimal" format. This is yet another data format. In BCD, the eight bits of a byte are interpreted to be two four bit decimal digits. Each four bit digit may have the hexadecimal values of 0-9. If there is some other bit configuration in the range

of A-F in either half of the bytes being added, the results will be unmeaningful. When the ADC is executed in Decimal mode, the two low-order digits are added and any decimal carry is added together with the two high-order digits. The carry bit will be set if there is a decimal carry from the addition of the high-order digits.

It is standard procedure to clear the carry bit before using the ADC instruction because the Carry bit is added into the result. This is a nice feature when you are doing multiple precision addition such as adding two 32 bit numbers together. The carry bit is the needed communication between the successive bytes of the addition.

Example of 32 bit multiple precision addition:

```
        LDX #3       4 BYTE ADDITION
        CLC
NEXT    LDA VAL1,X
        ADC VAL2,X
        STA VAL3,X
        DEX
        BPL NEXT
```

SBC - SuBtract with Carry. The specified byte in memory and the inverse of the Carry bit are subtracted from the A-reg, replacing the A-reg with the result. That is, (A-reg) = (A-reg) - (addr) - [1 - (Carry)]. The Carry bit is set if the contents of the A-reg are greater than or equal to the value being subtracted from it. The Carry bit is cleared if the value subtracted is less than the contents of the A-reg. In deciding whether the contents of the A-reg and the memory location are greater than or lesss than one another, the 6502 interprets the values as unsigned integers in the range of 0-255.

The carry bit is like an inverted borrow. If a borrow is required, the Carry bit is 0. If no borrow is required, the Carry is 1. The Negative and Zero bits are set based on the result of the subtraction. The Overflow bit is set if "two's complement overflow" ocurred. Valid addressing modes are the same as for the LDA instruction.

The normal subtraction procedure is to set the Carry bit before the SBC is executed. Using the SBC to compare two values requires testing the Minus bit after the subtraction is complete.

If the desire is to branch to PRGA if the contents of VAL1

are less than the contents of VAL2 the following program would be used:

```
           SEC
           LDA VAL1
           SBC VAL2
           BMI PRGA
OK         EQU ∂
```

The above routine works for unsigned numbers. If a comparison is being made of numbers which may have negative values, a more complex routine must be used. The possibility of "negative overflow" exists when subtracting numbers which are intended to represent values in the range of 0 to 127, -1 to -128. If we try to subtract 10 from -124, the result would be -134. This is out of the range of negative numbers. The 6502 lets us know that if this was a signed operation, the result had "negative overflow". This is done by setting the Overflow bit in the Processor Status Register. The value which ends up in the A-reg is 122 (256-134) in the above example. The unsigned equivalent of the -124 is 132 (256-124). And 132 - 10 = 122. This positive result gives a false indication of the relative magnitude of the two signed numbers. When working with signed numbers the following technique is necessary to make accurate comparisons of magnitude:

```
           SEC
           LDA VAL1
           SBC VAL2
           BVS CHKMI
           BMI PRGA
           BPL OK
CHKMI      BPL PRGA
OK         EQU ∂
```

Positive overflow can occur the same way. Suppose VAL1 has the value of 126 and VAL2 has the value of -4. The intention of the above program is to branch to PRGA if VAL1 is less than VAL2. 126 - (-4) = 130. 130 is out of the range of signed numbers (0-127). Positive overflow occured. The 6502 tells us this in the same way, by setting the Overflow bit. Does the program work for this case? 126

is not less than -4. The overflow bit was set by the SBC. The branch is taken to CHKMI where the Negative bit is tested. It is on because the value in the A-reg is greater than 127. It is 130. So the branch is not taken to PRGA. It works!

Compare Instructions.

CMP - CoMPare. The magnitude of the specified memory location is compared with the magnitude of the A-reg. The Zero and Negative bits are set as though a SBC had occurred. The A-reg is not modified by this instruction. The carry bit need not be pre-set or cleared before executing the instruction. The carry bit is set if the compare finds that the A-reg is greater than the value of the contents of the memory location. The Overflow bit is not affected by this instruction. It is therefore not possible to use the CMP to make magnitude comparisions of signed numbers. The description of the SBC instruction illustrates the technique for accomplishing this. Valid addressing modes are the same as for the LDA instruction.

CPX - ComPare the X-reg. The X-reg is compared to the contents of a specified memory location. This instruction functions exactly like the CMP instruction except the register being compared is the X-reg. The valid addressing modes are Absolute, Zero Page and Immediate.

CPY - ComPare the Y-reg. The Y-reg is compared with the contents of the specified memory location. This instruction functions exactly like the CMP and CPX instructions. Addressing modes are the same as CPX.

Impotent Instructions

NOP - NO OPeration. The amazing NOP instruction does absolutely nothing. It causes the 6502 to spin its electronic wheels for a few microseconds. It takes up one byte of memory.

The Tools

There are five software tools which make up the standard Kit. The Editor is used to create, modify and save assembly language programs to cassette or diskette. It performs certain error checking functions as well. The Assembler reads the assembly programs which were created with The Editor and The Decoder and produces a machine language program listing and machine language load modules on cassette or diskette. The Loader reads the load modules from cassette or diskette into memory where they may be RUN, single-stepped or SAVEd. The Monitor can single-step through any machine language program, displaying the various registers and status bits at every step. It also allows the easy examination and modification of memory locations. The Decoder does the opposite of The Assembler. It reads machine language programs from the computer's memory and converts them into assembly language which can then be saved to cassette or diskette for re-entry into The Editor and The Assembler. Listings of the assembly language program are also printed to the screen and/or the printer. With the extended-feature version of the Kit (see exchange offer at end of book), the Edtior and the Assembler have been combined into one program, ASMEDT. This version requires a minimum of 3K of expansion memory but is easier to use.

First Steps

Before attempting to use any of the software tools, backup copies should be made and the original tucked away for safe keeping. Safe keeping for any magnetic media means in a cool dry place away from sources of magnetic flux such as TV sets, loudspeakers, transformers, power supplys, exploding nuclear weapons etc.

To back up your programs, load them one at a time using the usual procedure: LOAD xxxxxx (or LOAD xxxxxx,8 for diskette) where xxxxxx is the program name. The names of the programs which you should find on the distributed media are: EDITOR, ASMBLR, LOADER, MONITOR, DECODER and ASMPRT. After loading each program, save it either on cassette or diskette as you please. Again, the usual procedure is followed: SAVE xxxxxx or SAVE xxxxxx,8. Be careful not to save the program onto the cassette you just read it in from as you will no doubt write over the next program on the tape. It would be a

good idea to save each program on its own cassette if you don't have a disk system. This will cut down on the time it takes to find and load the programs when you use them.

If you experience trouble loading any of the programs, be sure you have spelled the name right and that the tape is in the proper position. If LOAD ERRORs prevent you from loading the program or the program just doesn't seem to be on the media at all and you are sure your machines are in good working order, you should return the media to French Silk for a replacement. You have thirty days in which to find out if you have a problem and to return the faulty media. A replacement will be made at no charge.

The Editor

Load The Editor in the same way you did when you made your backup copy. Once it is loaded and the screen display says READY, type RUN. The screen will change colors and the copyright message will appear, followed by the prompt, "MAX?". This is the only time this prompt appears. You are being asked how large an assembly language program segment will you be wanting to create or load. It is only important to answer with a number at least one greater than the largest number of statements you expect to have in any segment. If you have a 5K VIC you can always answer 90. This means you will not be allowed to create a program segment larger than 89 statements. That is about the most you can get into the minimum VIC anyway. If you have some memory expansion, you can figure on getting about an extra 50 - 75 statements per K of expansion. So, with an 8K expander you can figure you can create and assemble a program segment of about 500 statements. Later on, we will get into breaking large programs into smaller workable segments.

For now, enter 90 regardless of how much expansion you have. Do this by keying "90 [R]". Note that at all times in this text we will use the "[R]" symbol to refer to the RETURN key.

You should now see "CPLDIMS" in the center of your screen and a "?" on the following line. You are being asked to select a choice from the menu of options. Briefly, the options are: C - create, P - print, L - Load, D - Delete, I - Insert, M - Modify, S - Save. (Note: with the Extended-Features version of the Kit, the additional menu option "A" for Assemble appears after the other editing options).

Since we are now going to create a program segment you should

Key "C [R]". The screen should now clear and " ?" " should appear on the second line of your display. This is the prompt for entering a line of a source program segment. It always appears in the same place on the screen. This is the "edit-window". When we get to modifying program segments we will see that lines to be modifyed will appear in this same edit-window.

Enter the following assembly language statement without hitting [R]:

LOW EQU7600

Be sure to leave a space after the label, LOW, and before the mnemonic, EQU. Be sure not to leave any space between the EQU and the operand, 7600. If you made errors in keying the above statement you may move the cursor to the error with the cursor control keys and correct it by typing over the error or by using the [delete] and [insert] keys. Until [R] is hit, changes may be made at will. Each statement entered in the edit-window may be at most 86 characters long. This is four complete screen lines.

Once you have the statement keyed just as shown, hit [R]. You should see the statement appear a few lines lower on the screen with the line number 1 on the left and with a space between the EQU and the 7600. Not having to key the space between the mnemonic and the operand is like an automatic tab feature and saves a keystroke. It can be confusing though, expecially if you are used to other assembler/editors which require the extra keystroke. Regardless, this is the format of every statement you will enter with The Editor. The label field is optional and if omitted, a single space must preceed the mnemonic. The first space after the mnemonic signifies the beginning of the comments. In this example no comments have been entered.

Now the edit-window should be blank except for the prompt (?").

It is time to enter the second statement of the program. Key the following:

MSG BYT'IT WORKS

The label field is "MSG". The mnemonic is the pseudo-op, "BYT", and the operand is "'IT WORKS". This would seem to be a

contradiction of what was just previouusly stated about the first space starting the comments. It is. This is the only exception to the rule. When the first character of the operand field of a BYT instruction is a " ' ", then no comments are allowed and the entire character string after the " ' " is considered the operand. This operand type is called a literal string.

When the statement is entered correctly, hit [R]. This statement should now appear after Statement 1 on the screen. It is line # 2 of the program being written. The edit-window is now open again.

Enter the statement:

SPG EQU658 SCREEN

Hit [R] when you've got it right. It appeared like the others down below the window, right? It should have. Now, the problem is: it is an incorrect statement. It should have been:

SPG EQU648 SCREEN

To fix it, we have to leave the mode of entering new statements. This is done by keying [R] with a blank line in the edit-window. Do that now. You should see a return of the original menu, followed by the " ? ". To fix a wrong line it is necessary to choose menu option "M" for modify. Do that and be sure to hit [R].

Now the prompt, "L,#?" should appear. This prompt is used for the M, D, P and I options. It means which Line number (L) and how many lines (#). For the Modify, the second number should always be a "1" since we can only modify one line at a time. Key: 3,1 [R] . The statement we just entered should now appear in the edit window. At this time it is possible to make any modification desired by using the cursor control keys, the [insert] and [delete] keys and keying over any characters which appear in the window. Move the cursor to the 5 and key a 4. You can hit [R] now if it looks like it should. There is no need to move the cursor to the end of the line before hitting [R]. Now the fixed version of line 3 should have replaced the erroneous one. The menu should have reappeared once again. We'll use the "P" option to see the altered statement.

The Print mode option allows you to list your segment to the screen. You may start anywhere in the program and print in blocks of

any size. Key "P" [R]. You will see the already familiar "L,#?" prompt. You are being asked where do you want to begin listing the program and how many lines do you want listed. We only have three lines so far so it would be reasonable to key 1,3 but for the sake of education, key 1,1 [R] instead. If you did that, you will see the first line was displayed and the menu reappeared. Now, if you key only [R] the listing will continue with the next block of # lines. It will print the same number of lines as you specified last time, i.e. 1. This is most convienient when you have a large program and wish to see a screenfull at a time. Answering 1,15 to the initial prompt will cause the first 15 lines to be displayed. And each additional time you hit [R] after that you will get another 15 lines. Hit [R] again and the last line will appear. Note that it was changed by your Modification process a while back. If you hit [R] any more, all you will get is the menu again.

We want to continue adding more statements to our program but it would be a mistake to try to continue by re-entering the Create mode. That causes the whole list of entered statements to be erased and the statement number to be reset to 1. DO NOT TYPE A "C" AT THIS TIME. Once the Create mode has been abandoned to fix a statement or to Print the segment, new statements may be added only with the Insert option

The "I" command, will cause us to re-enter the statement-input mode. Key "I" now to the Menu prompt. We won't bother to tell you to hit [R] after each entry any more because you already know that it is required.

If you entered "I", the "L,#?" should be staring you in the face again. For this mode, the right response is the Line after which you wish to insert new statements. The "#" is how many lines do you wish to insert. When all we want to do is to continue adding to the end of the program (like now) we enter the last line number of the program and the number 1 (3,1 in this case). The number of lines to be inserted is very important when you are inserting lines in the middle of the program. You could overwrite some of your program if you enter more than you say you will. You will have lines duplicated if you enter less. When adding to the end, however, we can add as many as we want and only specify "1" without having any problems.

You should enter "3,1" now and you will see that the lines of the program up to and including the line L will be listed. And the edit-window prompt will be back waiting for you to continue the

entering of your program. When entering program lines the most common error is to either forget to enter a space before the mnemonic or to enter one after it. If you get stopped with an ERR message it is probably because you broke one of those rules.

ERR 1 means either an invalid mnemonic was entered or more than one space or no space immediately preceded the mnemonic. ERR 2 means that either an addressing mode was used which is invalid for the mnemonic or that a space was entered immediately after the mnemonic.

The first three statements of the following program have already been entered. Now enter the remainder. You may experience occasional delays of a second or two after entering the input line before the prompt returns. Note that there is a 10 character input buffer which allows the fast typist to "get ahead".

```
LOW     EQU 7680
 MSG     BYT 'IT WORKS
 SPG     EQU 648 SCREEN
 ENDMSG  EQU *
 LENMSG  EQU ENDMSG-MSG
 ENTRY   LDY #LENMSG
         LDA #$08
         STA 36879 SCRN COLOR
         LDA #9*22+4
         STA ←0
         LDA SPG
         STA ←1
 LOOP    LDA MSG-1,Y
         AND #$BF
         STA (0),Y
         DEY
         BNE LOOP
         RTS  RETURN
 LAST    EQU ENTRY
```

The RTS may have caused you a problem. It takes no operand, so the "RETURN" must be separated from the RTS by a space since it is a comment.

Once you get all of these statements entered, exit the entry mode the same way we left the Create mode, i.e. [R] on a blank edit-window line. Now would be a good time to check the program for

accuracy.

It turns out there is a mistake in the way the program was written. For it to work correctly, statement 3 must be moved so that ENDMSG follows immediately after MSG. Also, an EQU LOW+LENMSG needs to be inserted just before statement 6.

To accomplish these modifications you need to first delete statement 4. You should have finished the insertion process of creating the above segment by hitting [R] with a blank line. The menu should be on the screen now. Give it a "D" for delete. The "L,#?" now may be responded to with a "4,1". This is to delete one line at line number 4. Once it is done, the menu will reappear.

Now to put it back where it belongs, you will need to do an Insert. You should know how to do this now. You will want to insert one line after line 2. When you get the " ?" ", re-key the ENDMSG statement followed by [R]. To get out of insert mode hit [R] with a blank edit-window.

One more thing to do. Insert the "EQU LOW" line after line 5. Now, re-list the entire program with the P option. It should look like this:

```
LOW     EQU 7680
MSG     BYT 'IT WORKS
ENDMSG  EQU *
SPG     EQU 648 SCREEN
LENMSG  EQU ENDMSG-MSG
        EQU LOW+LENMSG
ENTRY   LDY #LENMSG
        LDA #$08
        STA 36879 SCRN COLOR
        LDA #9*22+4
        STA +0
        LDA SPG
        STA +1
LOOP    LDA MSG-1,Y
        AND #$BF
        STA (0),Y
        DEY
        BNE LOOP
        RTS  RETURN
LAST    EQU ENTRY
```


If everything looks right, you can now save the segment you just created. If there are errors, use the modify or insert or delete commands to correct the problem before proceeding.

If you have a cassette version of the software tools, place a blank cassette in the recorder and make sure it is rewound. Give the menu an "S" for Save. If you have the diskette version, put any diskette with a little space left on it in the drive and key the "S". You will be asked the "NAME?" which you wish to assign to the file which you are about to create. If you have a disk, you must give it a name. With cassette, the name is recommended but optional. Respond with the name "TEST".

The source program segment will now be saved. With cassette, you will be asked if the recorder is "OFF?" before telling you to "PRESS PLAY AND RECORD". You need only hit return as this is just a pause to insure the recorder was not left on from the last input.

Once the recording is finished, the menu will reappear. Now the first session with The Editor is over. To proceed with the next step, assembling the program with The Assembler, either select the "A" option of the menu if you are using the extended-feature version or press the STOP and the RESTORE keys simultaneously and proceed to the next chapter.

The Tools - II

The Assembler - The program segments created with The Editor need to be translated into actual machine language. This is the function of The Assembler. It can read in and translate any number of consecutive source (assembly language) segments. For each segment read, it will generate a reprint of the original source statements, each followed by its machine language equivalent. It will print the machine language in either hexadecimal or decimal. It also optionally produces an object (machine language) output file which The Loader can than load into the specified memory locations of the computer. With the extended-features version of the Kit, the Assembler is a part of ASMEDT, the Assembler/Editor combination. The only functional difference with ASMBLR and the assembler portion of ASMEDT is that direct POKEing of the generated machine language into memory is an option of ASMEDT and it is not possible with ASMBLR. This feature, when used must not attempt to POKE machine language into the portion of memory which is used by the ASMEDT program itself, as this will cause self destruction of ASMEDT. The POKE option is selected by answering "Y" to the "POKE?" prompt.

Note that there are two version of The Assembler included on the distribution media. The second version, ASMPRT, outputs to the printer rather than the screen. To load The Assembler, have either the cassette or diskette with the saved copy of ASMBLR (or ASMPRT) in the appropriate device (recorder or disk drive) and key "LOAD ASMBLR" or "LOAD ASMBLR,8" if you have the diskette. When the READY. appears on the screen, key "RUN".

You will now get the same "MAX?" question as with The Editor. Here, again, you are being asked for the maximum segment size in number of assembly language statements. Respond with "90 [R]".

The next prompt will be "D/H?". Do you wish to see the machine language in decimal or hexadecimal? If you key "D [R]", the result will be decimal. Otherwise, any response including [R] will cause hex output. Key a "H [R]" this time.

Now the prompt should be "NAME?". Give it the name of the source segment you created with The Editor. In the previous chapter we created a source file named "TEST". That is what you should enter now. With cassette, if you hit just [R], the first file found on the cassette will be used. If the recorder was off, you will get the

"PRESS PLAY ON TAPE" message. Do so.

Now the file should be loading. Although this is a two-pass assembler, this is the only time the file actually has to be read into the computer. You can be sure the file has been found by listening to the recorder. There will be a slight pause every few seconds as the file is being processed. Somtimes, if cassette recorders are not in the best of health, files created with one program do not load very well with another program. File-creation has less built-in fault prevention than does program-creation and so files are more prone to load errors. The usual symptom of this problem with The Assembler is a "STRING TOO LONG" error. You can try to load it again or you might have to re-create it with The Editor. If it is a persistent problem, you might be well advised to have your recorder tuned up.

When the file has been successfully loaded, the next prompt appears. "OUTN?" is requesting a name to be assigned to the machine language output file. If you don't wish to create one at this time you must hit [R] only. Do that now.

Now the program will start to be listed on the screen. You should see the first ten lines of the program which was created with The Editor. After each line of source should be a line with the line number on the left, followed by the address of where the machine language will end up, followed by the actual machine language translation of the preceeding assembly language statement.

Ten such two-line statements will be followed by a single " ? ". The Assembler pauses every 10 statements to let you view the screen and to tell it when you are ready to have it continue. To get the next batch of 10 statements it is necessary to key anything but " / ". A " / " will cause the assembly to be prematurely ended and a return to the point of asking "D/H?". [R] is the normal response. Key that now.

The rest of the program should now be assembled and listed. It will be followed by the "END?" prompt. If you have all you need from The Assembler you may want to terminate the program by simply pressing RUN and RESTORE simultaneously. You may want to assemble another program segment at this time in which case hitting [R] will work. You may also re-assemble the same segment now without having to re-load the source file. That is what we will do now. Key [R] to the "END?" prompt.

We should be back at the "D/H?" question. This time, answer "D" for decimal. Now for the "NAME?" prompt, give a " / " followed by

[R]. This is the way to specify a re-assembly of the same file without having to re-load the source segment. The "OUTN?" prompt can now be answered with a file name to be assigned to the desired load segment. This time we will create one. Enter "TEST-L" followed by [R]. If you have the cassette version, you should get a "OFF?" prompt now. Like with The Editor, The Assembler is trying to protect you from your own neglect. If you had left the recorder in PLAY mode from the file input phase, The Assembler would have thought that the recorder was ready to record. This prompt is your cue to put the proper cassette in the recorder at the desired position on the tape. Once that is done, you may hit [R]. Now the PRESS PLAY AND RECORD ON TAPE message will appear and you should follow orders.

If you have done all of that, the recorder will start writing, the listing will begin to appear on the screen and this time the addresses and generated machine language will be in decimal instead of hex. Everything else is the same. You will notice the recorder or disk doing its thing every once in a while as load segments are being written out. At "END?" time the output file will close. You now have a file saved to tape or disk which will be loadable by The Loader.

There are two error messages which the Assembler can generate. "ERR 3" signifies a reference in the operand field to a label which has not been defined anywhere in the program. Following the "ERR 3", the offending label will be printed. After the label will appear the actual machine language erroneously generated. ERR 3 messages may also appear before the listing of the program begins during the file input stage. These messages flag EQU statements which have labels which have not been defined. See the Assembler specification chapter for more information on EQU statements.

"ERR 4" signifies a relative branch instruction with an operand expression which computes to an address which is out of range. Relative branches may be to addresses which are a maximum of 127 bytes beyond or 128 behind the address of the instruction immediately folowing the branch instruction. The same information follows the "ERR 4" message as the ERR 3.

The Loader - The capability to write, load and run a very large program is provided for even the smallest VIC-20 through the combined use of The Assembler and The Loader. Large programs may be broken

into segments which can fit into the memory space of The Assembler. Because The Assembler can create load segments, the space where the machine language program is designed to reside is not limited to that which is left over when The Assembler is in memory. That is, a failure of some assembler systems is that the assembler directly POKEs the machine language into memory. The memory where the assembler itself resides is therefore not available for the machine language program since the assembler would be self destructing as it was assembling. The solution is to have the assembler save the machine language program someplace where another program can load it into memory. This requires a loader program. If the loader is a short program it will allow the maximum possible space for the machine language.

It is also desireable for the loader to be able to input multiple load segments to better handle the large program. This, The Loader is and does.

The Loader is written in BASIC and because of the way BASIC handles string variables, it might be necessary to "protect" your machine language program from the string variables of BASIC. If you are using a VIC with 8K or less. Before loading the sample program of the previous chapters it will be necessary to reset the "TOP-OF-BASIC" and "TOP-OF-STRINGS" pointers in zero page. This should be done in the following way:

```
POKE52,7600/256:POKE51,7600-PEEK(52)*256:
POKE55,PEEK(51):POKE56,PEEK(52)
```

The Loader is loaded just like The Editor and The Assembler. The program name is "LOADER". When you RUN the program, the first prompt asked by The Loader is "NAME?", asking for the name of the load segment created with The Assembler. For the purpose of this tutorial, place the cassette with the tape created with The Assembler in the recorder (or the diskette in the disk drive) and key the name "TEST-L". The usual "PRESS PLAY ON TAPE" message will appear. Do it.

The tape or disk should now do its thing as the load module gets input and the appropriate memory locations get POKEd with the machine language program. As each load segment completes loading, The Loader will ask for the NAME of the next one. If there are no more to load, the response should be "/". For this demonstration we have only one segment. The second time the "NAME?" prompt appears, key "/".

At this point the machine language program created with The Editor, assembled with The Assembler and loaded with The Loader should be in memory ready to be executed.

The Loader will first prompt you with "SAVE?", asking if you would like to save the machine language program. This can be particularly useful for large multiple-segment programs. Saving it as one large program will enable you to re-load it like any BASIC program with the usual LOAD procedure. An interesting side benefit of The Loader is that you may use it without loading any Assembler-generated load segments. You may use only the SAVE feature by responding with a "/" to the first "NAME?" prompt.

If you choose the SAVE option, you will also be prompted for the low and high address of the seqment you wish to save. These must be entered in decimal. It is an unfortunate anomaly of the VIC that for cassette saves, the highest address which may be saved is 32767. This was presumably a feature to protect cartridges from being saved. However, it works fine for saving to diskettes. It would be a good exercise in understanding the workings of the Kernal to attempt to find the limiting portion of the operating system and to find a way around it. The Decoder would be very useful in such a pursuit.

You may answer "Y" to "SAVE", and 7600 to "LOW-ADR?" and 7680 to "HIGH ADR". Be sure to have a tape ready to record on, then answer "SAVETEST" to "NAME?". This will cause the creation of a file on cassette (or diskette if you have the disk version) with the name "SAVETEST". When you load it back in if it is on diskette you must key "LOAD SAVETEST,8,1" for it to load back into the same space you saved it from. If the file is on cassette, just "LOAD SAVETEST" will do.

Next, The Loader will ask you if you wish to "AUTO-SYS?". If you respond "Y". The program just loaded will execute via a SYS to the address specified in the last EQU in the program. If you wish to use this feature, the last EQU in your program must specify the entry point of your program. Our program had such an EQU so we can try now to execute the program. Key a "Y" to the "AUTO-SYS?" prompt.

If everything has gone well, you will see the produce of your programming labors now displayed in the middle of the screen. If for some reason this is not the case, it would be good to go back to The Editor, load the source segment with the menu option "L" and verify

that it is correct. Make any necessary changes and re-assemble and re-load the program. It should work.

The format of the load segment file which is created by The Assembler and gets input by the loader is quite simple. The records consist of three record types: address records, machine language records and an end-of-file record. Every time The Assembler processes an EQU statement it generates an address record. Address records cause The Loader to reset the address of where to next POKE the machine language. The second type of record is a machine language record. There is one record for every byte of machine language to be POKEd. As each machine language record is processed, it is POKEd into sequentially increasing memory locations until another address record or end of file record is encountered. The end-of-file record is a single "/".

Where to put the machine language program

There are several places where machine language programs may be designed to reside. These include 1) Inside a BASIC program 2) After BASIC 3) Below BASIC 4) In the cassette buffer ($33C-$3FB or 828-1019) 5) In expansion RAM in block 5 ($A000-$BF00 or 40960-49152). 6) Anyplace that The Loader itself does not reside.

Inside a BASIC program

To get a machine language program into a BASIC program so that it may be saved with the BASIC program and reloaded right along with it, a couple of techniques may be used. The first way is to load a BASIC program with several REM statements which take up space and will be overlaid by the machine language. The token for REM must not be overlaid but everything after it can be.

Record the values of locations 43-46, as you will need them later. Then load The Loader into memory starting at some address above the BASIC program (using the technique explained in Appendix F).

The Loader and the other BASIC program will both be in RAM now but in different locations. Now run The Loader to load the machine language program which was designed to fit in the space which the dummy REM statements created. This will cause the REMs to be overlaid. The best technique is to have the REM statements at the very front of

the program. This way, the machine language will not be shifted around every time statements are added to or deleted from the BASIC program.

It will be necessary to change some pointers within the text of your BASIC program if you overlay more than one REM. The last REM overlaid must be found in memory and its link address to the following statement must be used to replace the link address of the first REM. To do this you should know the format of BASIC programs in memory.

Locations 43,44 contain the address of the first BASIC statement in your program. At that address, will be a two-byte address of the next BASIC statement. This is the link address referred to. The end of the program is signified by a link address of 0,0. Immediately after the link address is the statement number, also in two-byte format. After that comes the compressed BASIC statement which is ended by a byte of $00.

Here is a one-line direct-mode program for finding the address of the start of any statement in BASIC:

```
I=43:FORJ=1TO10000:I=PEEK(I)+256*PEEK(I+1):
IFPEEK(I+2)+PEEK(I+3)*256<>xxxTHEN NEXT
```

I now contains the address of statement xxx.

Once the machine language has been loaded with The Loader it will be necessary to change the pointers at 43,44 and 45,46 back to what they were before The Loader was loaded. Now the Basic program will contain its machine language passenger and they may be all saved together.

Be cautioned that listing the program may cause funny characters, maybe even cause the screen to go blank or do other strange things.

There is a serious drawback to this system. If your program has a $00 anywhere in it, BASIC will get confused and modifications and SAVEing the BASIC program will cause the linkage pointers to be reset and will tend to mess up your machine langugage program.

One further caution. There is another value which BASIC has trouble with when within a BASIC statement. The value is 204 ($CC). Listing a BASIC program with this value within the program will cause a ?SYNTAX error when listing it. The program will still run however. And you may list the remainder of the program by typing "LIST 30-" for

example if the syntax error stopped the listing prior to statement 30. This might be considered a sort of protection feature, as it is likely to confuse the opposition (at least for awhile).

Another means of including the machine language program within the BASIC program is to run The Loader, poking the machine language into memory which you know to be after the address of the last line of BASIC code in your hybrid program. Now, load or enter the program. Next change the start-of-variables pointers at 45,46 to point to some place past the machine language. This method does not require the creation of REMs and the machine lanugage never gets printed but it does get saved and re-loaded with the BASIC program. The problem is that any modifications to the BASIC will shift your machine language around. This could be a problem if you have absolute addressing to locations within the program. Also, SYSs to entry points must be then changed too.

After Basic

The upper limit of the memory space which BASIC believes it has available for BASIC programs is maintained in an address pointer at locations 55,56. At 52,53 is another pointer which BASIC uses to set the highest memory address usable for string storage. BASIC starts storing strings at its highest available memory location and works back down. The free memory available in a BASIC program is the space between the high end of variable storage and the bottom of string storage. It is possible to save machine language program segments in a space which will not be molested by BASIC programs if you modify the two pointer sets at 52,53 and 55,56 so that they point to an address below the machine language. Then BASIC will not even know of the existence of that memory space and will not try to save anything there.

Cassette Buffer

There is a serious problem with putting the machine language in the cassette buffer with the cassette-based version of this kit. The Loader will need to use the buffer to read the load segments. It

cannot therefore POKE machine language into it without losing it as soon as it reads another record. If you have a diskette based system, there is no problem.

In expansion RAM block 5 ($A000 - $BFFF)

No protection from BASIC need be implememted when the program is up here. BASIC can't get to block 5. The only problem is that the cassette version of the SAVE won't work with programs in block 5.

Where The Loader isn't

When the machine language program is not designed to have a BASIC component the program may always be loaded with The Loader and run from there. Or The Loader may be incorporated into a BASIC program. In both of these cases the load modules created by The Assembler should follow on the same tape as The Loader (or the combination Loader/other BASIC program).

If the loaded machine language program is SAVEd with the Loader's built in SAVE feature as described above it may be simply loaded directly with a normal "LOAD".

A final note of caution. The loader takes some space in memory. If the space you wish to load your machine language program into overlaps the space consumed by The Loader, loading the program will destroy The Loader. If your Loader acts very strange, this is the probable reason. The solution is to select carefully where you wish to load The Loader so it won't be overwritten by the program it is loading.

Communicating between BASIC and Machine Language

As mentioned, one way of executing the program is through the AUTO-SYS option of the Loader. To get to the machine language segment from a BASIC program involves having either a SYS or a USR instruction in the BASIC program. Executing (running) a machine language program which has no BASIC component and which has been SAVEd with the SAVE option of the LOADER and subsequently LOADed with the normal LOAD command invoves typing a SYS or USR instruction in direct mode.

The format of the SYS statement is "SYS addr", where addr is

the address of the first statement in the program to be executed. This will cause a jump to the specified address. Data may be passed to the machine language program by POKEing it into some memory locations where the machine language program has been programmed to find it. It may also be passed directly to the A-reg, X-reg, Y-reg and P-reg by POKEing it directly into locations 780-783 respectively. The BASIC SYS command automatically loads these four registers with the values found in these locations before passing control to your machine language program. It also returns to your BASIC program with 780-783 containing the values of the above registers as your machine language program left them.

The USR instruction format is "X=USR(A)". This instruction is used in place of the SYS when you wish to pass a real variable (floating point number which contains a fractional component) to the machine language program. The value of BASIC variable "A" is loaded into the floating point accumulator # 1 (see chapter 13 for a full explanation) before passing control to the machine language program. Upon returning to the BASIC program, the value which the machine language program leaves in in the FAC#1 will be assigned to the BASIC variable "X". The address of the entry point of the machine language program must be stored in locations 1 and 2 prior to executing the USR instruction. For example, if the entry point of the program is 7680, you must have the following BASIC statement preceeding the USR instruction: POKE2,7680/256: POKE1,7680-PEEK(2)*256.

With both the SYS and USR instructions, the way to return to the BASIC statement immediately following the SYS or USR is by the RTS instruction in the machine language program.

The Tools - III

The Monitor - The Monitor provides the capability of running a machine language program one instrucition at a time. As each instruction is executed, the internal registers of the VIC's 6502 microprocessor are displayed, including the individual bits of the status register. The instruction to be executed is also displayed, both in machine language and in assembly language. The capability is provided to bypass the execution of any instruction and to display and modify any memory location while in the process of running the program. The menu of choices is displayed at every step of the execution.

For the purpose of demonstrating the features of The Monitor, you should have loaded into memory the program created, assembled, loaded and run in the previous chapters. If The Loader just loaded the machine language program "TEST-L" everything should be ready to go. Otherwise you will need to go back and load it again with the instructions in the previous chapter.

Like with The Loader, if you are using a VIC with 8K or less it will be necessary to modify the TOP-OF-STRINGS and TOP-OF-BASIC pointers at 51,52 and 55,56 before loading The Monitor. See the previous chapter for information. If the pointers have not been restored since the previous modification of these pointers it will not be necessary to do it again. By the same token, subsequent programs which are loaded without resetting these pointers may find themselves "OUT-OF-MEMORY" because of the change.

Loading The Monitor is by the same process as with all programs in the Kit. The name of the program on the distribution media is "MONITOR". It may sometimes be necessary to load The Monitor into some part of memory other than where BASIC programs normally load. If the program you wish to monitor is in the normal BASIC space you may want to load it into some other space as per instructions in Appemdix F.

For the purpose of this tutorial, The Monitor may be loaded in the usual space without modifying the pointers. Once it is loaded, Key "RUN". The menu should now appear.

"S" is the menu option to (S)et the address of the next instruction to be executed. It will be followed by the "ADR" prompt,

asking for the starting address. Valid replys are decimal numbers in the range of 0-65535 and hexadecimal numbers in the range of $0000-$FFFF.

"M" causes The Monitor to enter the (M)emory display/modify mode. This is also followed by the "ADR" prompt, requesting the starting address of veiwing/modifying.

"B" will cause the instruction about to be executed to be (B)ypassed. Rather than execute the displayed instruction, the instruction following the displayed one will be displayed.

Hitting [R] with no menu selection will cause the current instruction to be executed and the next instruction in the program flow to be displayed.

Key a "S" to cause single stepping and answer the "ADR" prompt with 7608, the address of the entry point of the program. The first instruction of our program should now appear on the screen.

As each instruction is about to be executed, it is displayed in both machine language and in assembly language. The status register is broken into its individual bits (N = negative, V = overflow, B = break mode, D = Decimal mode, Z = Zero, I = interrupts inhibited, C = carry). The other registers displayed are the A-reg, the X-reg, the Y-reg, and the Stack Pointer (SP). Upon execution of each instruction, the registers are loaded from these save areas in memory:

X - 251 Y - 252 A - 253 P - 254

If you wish to modify or pre-initialize any of the registers at any time you may do so by entering the Memory-Modify mode and modifying the above locations.

A "software" stack is substituted for the hardware stack in page 1. Its maximum depth is pre-set at 20 bytes. While this will be sufficient for the vast majority of your testing, you may increase it if necessary by setting the new maximum depth in the SD variable in statement 1. If, while single-stepping through some program, you should execute a RTS or PLA or PLP without first having pushed somthing onto the stack with a JSR or PHA or PHP, a stack underflow will occur. A TXS instruction setting the SP to some out-of-range

value will also cause stack underflow. Overflows are caused by repetitive PHA's PHP's or JSR's without corresponding PLA's PLP'S or RTS's until the maximum stack depth has been exceeded. In the event of underflows and overflows, a "STK ERR" message will be displayed. Execution may continue if desired but results are likely to be unexpected if your monitored program is expecting to find some significant information in the stack (like a return address).

Certain machine language programs are written to modify the Stack directly by storing data in the high end of page 1. Executing these instructions with The Monitor will not cause the desired stack modification effect. In fact, it is quite likely that The Monitor will actually crash upon the execution of such instructions. Since The Monitor is written partially in BASIC, any instructions which modify the BASIC vectors or other information vital to the running of BASIC programs may cause unwanted results.

As The Monitor single steps through a machine language program, it checks each op-code encountered for validity. If an invalid op-code is encounterd, the message "OP-CODE = xxx" (where xxx is the encountered op-code) will appear where the mnemonic would otherwise appear. The Monitor will not try to execute invalid op-codes. Nor will it try to execute BRK or RTI instructions. All of these will be automatically bypassed.

If you have created and loaded the sample program and have loaded and run The Monitor and have selected menu option 5 and have answered 7608 to the address prompt, you will now see on the screen the assembly language statement "LDY #8" followed by a line with the address 7608 and the machine language equivalent of the above assembly language statement: 160 8.

Following the machine language will be the display of the registers prior to the execution of the displayed instruction. They will have no particular significance at this point because they were never initialized. Note, however, the value of the Y-reg because after executing the instruction it should change. The Zero flag, if it is a one now should also change as a result of the execution of the instruction. To execute the instruction, hit return. Do that now.

If all is well, the next instruction of the sample program will be displayed and the first instruction will have been executed. You may verify that by checking the Y-reg. It should be of value 8 now. The zero bit should be a 0 because the result of loading the Y-

reg is not zero. Note the disassembled statement is not identical to the assembly language statement we wrote. It is equivalent but not identical. We wrote the statement: LDA #9*22+4 . The displayed statement should be: LDA #202. This discrepancy occurs because The Monitor can't tell what went into The Assembler, only what came out and it does the best it can in reconstructing a valid assembly language statement from the machine language it has to work with. It generates all operand field values as a simple decimal value. When this instruction is executed we should see the A-reg's value reflect the LDA operation. Note also that the number 202 is greater than 127 and, as such, the high-order bit is on. Consequently, the Negative flag should be turned on by the execution of this instruction. Hit [R] and see.

The next instruction which is now up for execution will store the A-reg in location 0. None of the status flags are affected by this instruction so we should see no change when we execute it. Do it.

This would be a good time to look at the Memory Display/Modify mode. Rather than hit [R] at this time, key "M [R]". You are now in the M-mode.

Upon entering Memory mode, the address where you left off in single-stepping will be saved and the "ADR" prompt will be displayed. You may enter the first address you wish to examine or modify. The address may be in the range 0-65535 or $0000-$FFFF. The value of the data stored at the requested address will be displayed and the prompt "VAL?" will appear. You may do one of three things. You may eXit the Memory mode by keying "X". You may modify the displayed location by keying a value in the range 0-255 or $00-$FF. You may continue viewing the next sequential memory locations by hittng [R].

To look at location 0, key a "0 [R]". The address (0) will be displayed followed by the contents of the specified location. In this case it should be 202 because that is the value we just stored there.

Following the information line will be the question "VAL?".

You may now change the contents of location 0 if you wish by keying some new value. The next location will now be displayed, memory location 1. Note its contents and modify them if you wish.

To just scan through memory, simply continue to hit [R] each time a value is displayed. When you wish to return to the main menu, key "X" for exit. Once this has been done, the SMB menu should reappear.

If you modified location 0 you should now re-enter M-mode by entering the "M" option, and address 0 again. When the value you stuck in 0 is displayed, change it back to 202, hit [R], and when location 1 is displayed, hit "X" to get out again.

Now to return to the program we were single-stepping through, simply hit [R]. We could have started single-stepping at any location we wanted to by entering option "S" and the new address where we wanted to go to.

The next instruction in the program is now displayed. It is a LDA instruction. Note the registers still have the same contents as before the M-mode excursion. The A-reg will now be loaded from location 648. Location 648 contains the operating system's information about which page of memory the TV screen's data is kept in. When putting data into the screen's memory space, it will automatically appear on the screen of your TV. More about this in later chapters.

This program is setting up a vector in locations 0 and 1 of zero page to address the 4th column of the 9th line of the screen. A loop in the program will sequentially move characters from a data area in the program (MSG) to the screen memory where they will appear on the TV.

Hit [R] and see the next instruction which is a Store of the A-reg to location 1. Hit [R] again. The next instruction is the beginning of a loop. It loads the A-reg with the character at address 7599 plus the contents of the Y-reg. The Y-reg should still have the value of 8 so the address where data is being picked up from is 7599 + 8 or 7607. Go into M-mode and see what is there. It should be the ASCII value of the letter S, or 83. This is the last character in MSG in the original assembly language program.

Go back to the program now by eXiting and responding to the menu with a [R]. The instruction will be executed. The A-reg should have the value 83 and the AND instruction is up now. Its function is to convert the character in the A-reg from ASCII to VIC screen format. The screen format for the letters A-Z have the 6-bit off while the

ASCII characters have it on. The AND of the A-reg with the hex value $BF will accomplish that. See Appendix D of The Programmers Reference Guide for more information.

Execute the instruction and notice what happens to the A-reg. It now has the value of 19. The 6-bit has a bit value of 64 and turning it off should reduce the value of the A-reg by 64. Which it did. The next instruction to be executed will store this value at the location computed from the sum of the contents of the Y-reg and the address vector in locations 0 and 1. If you take the value stored at location 1 (which is the page # of screen memory) and multiply it by 256 and add the value stored in location 0 (202 or 9*22+4), you will get the base address to which the Y-reg is added. All vectors work the same way: add the contents of the first byte of the vector to 256 times the second byte to get the address being referenced.

After executing this instruction with The Monitor, it may actually be difficult to find the value where the program just put it. This is because the screen memory is modified by every program which writes anything on the screen. The Monitor writes quite a bit to the screen, so the screen memory is altered about as fast as our little program can put somthing there. This is one of the unfortunate facts of life in debugging programs. If we know in advance to expect this kind of problem we may not end up quite so confused when it happens.

Execute the next instruction, the DEY. The Y-reg should now have the value of 7. What is the contents of the Z-flag? It should be zero because the DEY did not cause a result of zero. Execute the next instruction. The BNE tests the Z-flag. It branches if the Z-flag is not a one. It is not, so the branch should take place. Before executing the BNE, check the address of the machine language instruction. Now execute the BNE. The new address of the machine language instruction is less. In fact, the new instruction ready to be executed is the LDA back up at the top of the loop. The branch was taken.

It would be good now to step through the loop quickly, watching the Y-reg get smaller and smaller each time through. When it gets to one, watch what happens at the DEY. The Zero-flag should go from 0 to 1 when the Y-reg goes from 1 to 0. The BNE then should not take the branch. The address of the instruction up after the BNE should be two greater than the address of the BNE itself. At this time, all eight bytes of MSG will have been copied into screen memory (only to be obliterated by The Monitor).

When this program was run by the Loader, the machine language program had control for the entire time the characters were being copied to the screen. The Loader did no further screen activity so you were able to see them appear on your screen then and not now.

The last instruction in the program is the RTS. This is the way a program entered by a BASIC SYS returns to BASIC. If executed now, it will try to return through a stack which does not have any return address to return through. The Monitor warns you that this has happened. Try executing it and see.

The Monitor is a very powerful tool for both debugging and for learning how the instruction set works. You can use it to single-step any machine language program including the BASIC operating system.

The Tools - IV

The Decoder The opposite of The Assembler, The Decoder turns machine language back into assembly language. The format of the assembly language program is identical to that which The Assembler and The Editor accept as input. The Decoder produces the program listing to the screen, optionally to the printer and, also optionally, to either the cassette or the diskette depending on the version of the Development Kit you have.

The Decoder produces all address and operand fields in decimal. It does not produce labels. The program created by The Decoder will resemble the original assembly language program which created the machine language but it will not be identical. This is due in part to the lack of label generation. It is also due to the great flexibility available to the assembly language programmer in specifying address expressions. The Decoder can't know what the original expression was. It only sees the end result, which is an address or immediate data value in binary. It converts all such information to a simple decimal value.

The Decoder is loaded and run in the same manner as the other tools. The name of the program on the magnetic media is DECODER. The Decoder, like The Monitor and The Loader may be loaded into any available RAM. The first prompt you receive when you run The Decoder is "MAX?". This prompt is asking for the maximum segment size you might wish to create for eventual input to The Editor or The Assembler. If you have no intention of using The Decoder for this purpose, you may hit [R]. If you do create a segment on cassete or diskette to be assembled and/or modified by The Editor and The Assembler, respond with the maximum number of assembly language statements you wish to automatically create with The Decoder. If you select the option to create an output file, the program will automatically stop decoding upon reaching the entered maximum value.

The next prompt, "PRINTOUT?", allows you the option of creating printed output. If you reply with a "Y" you will automatically receive a printed listing of the decoded program on your VIC-1515 printer (or equivalent). Any other response including just [R] will suppress printing.

Next, you are asked if you want to create an output file. If you answer "Y" it will cause a copy of the decoded Assembly text to be automatically SAVEd to diskette or cassette in the special format

required by The Assembler and The Editor. Any other response will suppress SAVEing. If you do so choose, you will be prompted for the "NAME?" of the segment to be saved. You may respond with the usual legal file names or, in the case of cassette, no name. In the case of the diskette, if you key a name already in existence on the diskette, The Decoder will replace the old version with the new.

You must tell The Decoder where to begin decoding by answering the "START ADDR?" prompt with either a decimal number in the range of 0-65535 or a hex number in the range $0000 - $FFFF.
If you respond by hitting [R] without any entered value, The Decoder will begin decoding at the address where it left off. If this is the first decoding since you loaded the program, hitting just [R] will cause it to start at address 0.

You must tell The Decoder where to cease decoding by your response to "END ADDR?". Your response may either be an address or a specific op-code. If you specify an address, it must be in the range of 0-65535 or $0000-$FFFF. If you specify an op-code you must preceed the op-code with an "#". The op-code may be specified in either decimal or hex ("$" preceeding the op-code and after the "#"). Specifying an op-code will cause The Decoder to check each op-code as it decodes the machine language and to stop decoding after the first occurrance of the specified op-code. Hitting simply [R] will cause the most recently entered response (either op-code or address) to this prompt to remain the active stopping signal. In other words, if you decode several subroutines you only have to give the #$60 response once the first time, hitting [R] the following times.

The first statement output will always be an EQU statement which will have as its operand the decimal value of the Start Address. Each statement, as it is generated, will display the decimal value of the address at which the decoded instruction resides in memory. The address will be followed by the mnemonic (such as JSR or BNE), followed by the operand field. If an invalid op-code is encountered by The Decoder where it is expecting to find an a valid 6502 op-code, a BYT instruction will be generated having the decimal value of the encountered byte as its operand.

The operand field will indicate the addressing mode with the following conventions: "#" as the first character of the field indicates Immediate mode. "←" will preceed the operand field of zero page addressing and ZP,X and ZP,Y. Addresses enclosed in (...) are designated indirect addressing modes. The suffixes of ,X and ,Y

imply indexed by reg-X and reg-Y respectively. All address values are expressed in decimal.

After the Decoder reaches the designated end of the segment, whether by hitting the MAX value in the case of a SAVE, or by reaching the high address or by encountering the specified stopping op-code, The Decoder will return to the prompt for "PRINTOUT?". You may now decode another section of memory, selecting anew the options (PRINTOUT, SAVE, START ADR, END ADR)

Decoding cartridges is not a problem unless the cartridge is an auto-start cartridge. Then it is generally necessary to have a hardware assist. Many expansion chassis come with a set of switches which enable and disable the various slots on the cartridge. With this feature, the auto-start cartridge may be effectively decoded. It is necessary to have the cartridge in a switchable slot of the expansion chassis with the slot disabled when the power-on or reset process occurs. Once that has happened, the slot must be enabled by throwing the switch. Now the cartridge is in the addressable space and you still have control of the machine. Load The Decoder and specify address $A000 as the starting address and decode away.

To decode cassette or diskette based programs which start upon loading, it is necessary to have a reset button. Most good expansion chassis also have one of these. Load the program, hit the reset, load The Decoder into some space not used by the auto-start program, and decode. The reset function will destroy a few bytes of data at the START-OF-BASIC. If you have an expansion chassis and some expansion RAM, it is possible to change the START-OF-BASIC by switching the expansion RAM either on or off. With expansion, the start address for BASIC is 4609. With no expansion, the address is 4097.

The Decoder is a powerful learning tool. With it you may see how other well written programs do what they do. The coding techniques of others can become incorporated in your own bag of tricks. They will provide you a valuable base for building your skills inventory. It can be particularly helpful to examine the operating system to more fully understand how the VIC was designed and how its internal features are utilized by the systems programmers who wrote BASIC.

Specifications for Assembly Language

Labels

The label field, if used, is the first field of the assembly statement. It is not a required field. If used, it must start with a letter (A-Z) and may be of any length. The single character "A" should not be used, as it will be confused with the "A" of the Accumulator addressing mode when appearing in an operand field. Other characters which should be avoided are the seven algebraic operators and the quote ("). Labels of more than four characters will cause the assembly listing to be somewhat less neat appearing but work just fine. The longer your labels are, however, the less statements that can fit into a segment.

Mnemonics

Standard mnemonics

The mnemonic is the English-like code which gets translated by The Assember into the machine language op-code. A discussion of the mnemonics is found in chapter 5. See appendix A for a complete list of all valid mnemonics and their legal addressing modes.

The EQU pseudo-op

The Assembler comes equipped with two special mnemonics, EQU and BYT. These do not generate processor-executable instructions as do the standard mnemonics. The BYT pseudo-op is covered in the next paragraph. The EQU mnemonic is an instruction to The Assembler rather than an instruction to the machine. It has two functions. The first is to set the location counter. This is a function sometimes left to a separate pseudo-op such as ORG in other assemblers. The second function is to equate a label with an address. The format of the EQU instruction is:

```
LABEL EQU addr-expression
```

The label field is optional. There is one exception to the

general rules for address expressions for the EQU instruction. The expression may not reference labels which do not precede the EQU instruction in the segment. This is true only for the EQU instruction and is the only limitation on it. Any program references to the label on the EQU instruction will refer to the address expressed in the operand field.

The last EQU in a set of source segments generates load segment code which The Loader interprets as the entry address of the program. The Loader will ask if you wish to AUTO-SYS? at the end of loading load segments. If you answer "Y", it will SYS to the address expressed in this EQU.

The BYT pseudo-op

The BYT instruction, unlike the EQU instruction, does cause machine language code to be generated by The Assembler. It is not generally used to generate executable code like the standard mnemonics. Its function is to provide a means of causing data to be stored in memory. The data generated by the BYT instruction may be specified in several ways. Depending on the first character of the operand field, the BYT instruction may specify hexadecimal strings or ASCII strings or address constants or single byte values of hex, decimal or ASCII.

Hexadecimal strings

If the first character of the operand field is "$" then a hex string will be generated. All characters following the "$" should be hex digits, 0-9 A-F. There may be any number of such digits. The Assembler will create one byte of data for each pair of digits. If there are an odd number of digits, The Assembler will append a "0" on the left of the string.

Literal text strings

If the first character of the operand field is a " ' ", all following characters will be translated to the Commodore ASCII value of the characters. Each text character in the operand will generate one byte of data. All the characters which may be entered from the

keyboard with the exception of the quote (") are legal. The BYT literal instruction is the only instruction which cannot have a comment field. Comments would be interpreted as part of the literal text.

Data constants

A "#" in the first position of the operand field causes the expression which follows to be evaluated by the rules of address expression evaluation. The single byte which is generated is the low-order byte of the resultant two-byte evaluation.

Address constants

If the first character of the operand does not meet any of the above criteria then the operand is evaluated as an address expression as defined later. The two bytes of data which are generated are in the 6502 address format, with the low-order byte preceding the high-order byte. It should be noted that a BYT instruction such as:

```
BYT $3CC
```

will not generate an address constant in the low-high format. It is a hexstring and will generate 03 CC. To get an address constant it would be required to write an instruction such as:

```
BYT 0+$3CC
```

which would generate CC 03, as expected.
The following will also generate the same address expressiom:

```
MSG EQU $3CC
    BYT MSG
```

The operand field

The operand field follows immediately after the mnemonic field

and specifies to The Assembler the address of the data to be accessed and the mode by which it will be addressed. There are actually 13 distinct addressing modes by which the location of data is specified. The format of the operand field determines which mode will be used and therefore exactly which op-code will be generated and how many bytes of address data will be generated. Not all instructions may use the same set of addressing modes. Appendix A specifies the valid addressing modes for each instruction or mnemonic.

Address expressions

One of the great strengths of this assembler is the ability to create address expressions of great complexity (or simplicity) with ease and flexibility. The term "address expression" is meant to include the "immediate" character and BYT data constants as well as actual memory locations.

Terms

There are five different kinds of terms which may be algebraicly combined in an address expression. Each has its own distinguishing format to identify it.

Decimal format

Any term in an address expression which begins with 0-9 or a "." will be interpreted as a decimal term. Decimal terms may be integers or may contain a decimal point and a fractional component. There is no practical upper or lower limit to their magnitude.

Hexadecimal format

Hexadecimal terms are those which have a "$" as the first character. All following characters, up till the next operator or the end of the expression, must be 0-9, A-F.

Literal format

Any term in an address expression which begins with the

character " ' " will be interpreted as a literal. That is, the value assigned to that term will be the ASCII value of the first character to the right of the " ' ". Any characters following the first character following the quote will be ignored.

Symbolic label

Any term which does not start with a "$" or a "'" or a "@" or a "." or a number (0-9) will be interpreted as a symbolic label. The Assembler will search the entire program looking for a match on the label field. If none is found, ERR 3 will be generated. If a match is found, the term will be assigned the value of the address associated with the label. Labels may be of any length.

Location counter

The location counter is The Assembler's equivalent of the program counter. The location counter has the value of the current address assigned to the first byte of an instruction. It is the address at which the instruction will reside once The Loader POKES the load segment into memory. This assembler uses the "@" symbol to signify the location counter. Most other assemblers use the "*" symbol for this function. However, the "*" is interpreted by The Assembler as a multiplication operator. The "@" symbol seems a logical choice, being the "at" sign and signifying where we are "at" in memory. Any term which has "@" as its first character will have the value of the location counter. Any following characters within the term, should they exist, willbe ignored.

Algebraic operators

The various terms of the address expression may be combined algebraicly by the following operators: + - * / ^ & % . As the evaluation proceeds from left to right, each term is added to, subtracted from, multiplied by, etc. the result of the evaluation of that portion of the expression to the left of the operator, giving a new current evaluation. The fractional portion of the result of any operation will be carried into the next operation. The final evaluation of the expression will truncate any fractional components and will convert negative numbers into sixteen bit two's complement

values.

Addition " + "

The addition operator causes the term immediately following the "+" to be added to the result of the evaluation of the portion of the expression to the left of the "+".

Subtraction " - "

The subtraction operator causes the term to the right of the minus sign to be subtracted from the expression to the left of the minus.

Multiplication " * "

The multiplication operator causes the expression to the left of the " * " to be multiplied by the term to the right of the " * ".

Division " / "

The division operator causes the expression to the left of the " / " to be divided by the term to the right of the " / ".

Exponentiation " ^ "

The power operator causes the expression to the left of the " ^ " to be raised to the power of the term to the right of the " ^ ". Fractional powers may be employed with decimal terms. It is therefore ossible to do such things as take square roots with expressions such as:

X ^ .5

Logical AND " & "

The logical AND operator causes the result of the evaluation of the expression to the left of the "&" to be logically ANDed with the term to the right of the "&". The logical AND operation compares the two terms of the operation bit by bit, giving a result with a bit set on in every bit position where both terms have a bit on. Its main

use is to force bits off in certain desired bit positions, while retaining the status quo in all other positions.

Both the AND and the OR instruction have a restriction on the range of the value of both the expression to the left of the operator and the term to the right of the operator. If either is greater than 32767. The Assembler will stop with an ILLEGAL QUANTITY ERROR. This means that the range of address expressions which may be operated upon by these operators must be less than $8000. This restriction occurs because of the way the AND and OR operations are handled by the VIC. It is possible to partially defeat this restriction. By specifying an expression such as:

X-65536&Y

the value of X may be in the range of 32768 to 65536 ($8000 - $FFFF). Y must still be in the range of 0 to 32767. This results in ANDing the full 16 bit values of X and Y. If both X and Y are greater than 32767, there is no provision for successfully ANDing or ORing the two values n an address expression. Remember that X&Y = Y&X and X%Y = Y%X.

Logical OR " % "

The logical OR operator causes the result of the evaluation of the expression to the left of the "%" to be logically OR'ed with the term to the right of the "%" . The logical OR operation compares the two terms of the operation bit by bit, giving a result with a bit set on in every bit position where either term has a bit on. Its main use is to force a bit on in certain desired bit positions, while retaining the status quo in all other positions. See the previous section for a description of limitations upon the use of this instruction.

Expression evaluation

As has been indicated in previous sections, several terms may be combined into an algebraic expression, the eventual evaluation of which will result in the address specification. There are several features of the evaluation algorithm which must be understood for proper use of the algebraic capability. First, the order of evaluation is not like BASIC. Here, the expression is evaluated from

left to right, regardless of what operators appear in the expression. For example, the expression:

LA&1+GH%$C/IJ

would be evaluated in the following way: The address of LA would be AND'ed with the value 1. The result would be added to the address of GH. That result would be OR'ed with $0C. The result of that operation would be divided by the address of IJ. The final massage of the expression converts negative expression values to a two's complement value by the addition of 65536 to the negative value. By way of example:

-1 = $FFFF, -2 = $FFFE, -256 = $FF00, and -257 = $FEFF.

Finally, expressions which result in values greater than $FFFF (65536) are made modulo 65536. For example, 65538 becomes 2, 65539 becomes 3, etc.

Complex equations

For those who wish to use more complex equations than can be handled by expressions which are evaluated strictly left to right, it is possible to accommodate them by a series of EQU's which themselves are expressions. For example, to represent an equation such as:

(B + C - (D / E)) * (F & G)

you could write the following code:

```
DE   EQU D/E
FE   EQU F&G
AB   EQU B+C-DE*FE
```

Much more complex expressions may be represented in a similar fashion.

The comment field

Comments are entered on the statement line by skipping at

least one space after the operand field before keying the comment. Comments are not allowed on BYT instructions which define literal strings. Comments are useful for documentation but do consume memory and so cause the number of statements per segment to be smaller.

Writing multiple segment programs

To write programs which are too long to fit in a single segment, there is only one requirement. All references to labels which occur in some other segment must have an EQU included in the segments which reference them. It is therefore necessary to assemble the segment with the naturally occurring label before a final assembly of the referencing segments.

Because The Assembler keeps track of the location counter between segment, it is not necessary to include an EQU at the beginning of each subsequent segment to set the location counter. It will, of course be necessary to set the location counter at the beginning of the first segment. The default initial value of the location counter is 0.

Graphics on the VIC-20

These next chapters will attempt to expand upon the information found in the VIC-20 Programmer's Reference Guide (PRG). The PRG should be considered an indispensible reference tool. In it you will find complete descriptions of the various special function integrated circuits which make the VIC the powerful computer it is. Also included is information on the VIC's memory organization and the "Kernal" routines and how to use them.

This chapter will provide the machine language programmer's perspective on graphics generation. Joy stick and paddle usage and sound generation will be covered in the following chapter. The last chapter will cover some of the internal programs contained in the VIC's ROM and the means by which you can make use of them.

Custom Characters

The Video Interface Chip (VIC), after which the VIC was named, is the electronic machine, the integrated circuit within the VIC which, among other things, causes patterns to be displayed on your video screen. It is also known as the 6560, or on some machines, the 6561. The VIC chip is connected to the 6502 and the RAM and ROM of the VIC via the address, data and control busses. Its registers are wired directly to the address bus and may be written into and read from by any program running on the 6502. The registers occupy the addresses $9000 (36864) to $900F (36879).

You have no doubt noticed that each character which is displayed on your screen is a composite of up to 64 dots arranged in an 8 by 8 block. The information which describes the characteristic dot pattern of each character is stored in the VIC's ROM.

A certain section of RAM is reserved for use by the VIC as "screen" memory. There is one byte of screen memory reserved for every character position on the screen. Since there are 22 columns and 23 rows there are 506 bytes of screen memory. When data is stored into screen memory it gets translated into dot patterns by the VIC chip. The VIC chip interprets each byte in screen memory as a code which it has to translate to find the proper pattern of dots to put on the screen in the corresponding screen position.

The BASIC "PRINT" statement causes "screen codes" to be

stored in screen memory. It is possible also to POKE screen codes into screen memory. Machine language programs may store screen codes into screen memory. A code stored in the first position of screen memory represents the character to be displayed in the upper-left corner of the screen. The 22nd position of screen memory contains the code which represents the character which gets displayed in the upper right corner of the screen. The 506th position of screen memory contains the code for the character in the lower right corner of the screen.

The VIC chip automatically and continually scans the screen memory, getting one screen code per screen location and translates the screen code into an eight by eight dot pattern which it then sends to the video output port to be displayed. In appendix D of the PRG there is a table of screen codes and the corresponding characters which get displayed. These are the standard characters which are burned into the VIC's ROM.

The screen codes are actually an index into an 8 x 8 pattern table. Screen codes may have any value in the range of 0-255. There are therefore 256 different characters which may be displayed on a VIC-20 screen at any one time and 256 entries in the pattern table.

You are probably aware that when you hold the Commodore and the Shift keys down simultaneously, the characters which appear on the screen are from a second character set. One character set contains upper and lower case characters and the other contains upper case and the graphics characters. There are two sets of 8 x 8 patterns stored in ROM and the VIC chip can be "switched" between the sets. In fact, the VIC chip can be switched between several different tables of 8 x 8 patterns, two stored in ROM and others created by the programmer and stored in RAM.

Creating a pattern table is how custom characters and bit image graphics is accomplished. The pattern tables consist of 256 sets of patterns. Each pattern set has eight bytes. The 8 x 8 patterns take eight bytes of data to describe each pattern. The first byte represents the top row of dots in the character. The second byte represents the second from the top and so forth. A bit turned on in any of the bytes will cause the corresponding dot on the screen to be illuminated. The screen dots are also called pixels which comes from "picture elements".

The standard character set is stored in ROM starting at 32768. The screen code "0" refers to the first 8-byte pattern, that of the

character "@". The second character in the standard character ROM starts at 32776 and is the pattern for the letter "A" (screen code 1). Every eight bytes, another pattern is stored. This goes on for 256 8-byte patterns. The patterns for screen codes 128-255 are the "reverse" of the first 128 patterns. That is, where a bit is on in one, it is off in the other.

Screen memory starts at either address 7680 ($1E00) if you have 8K or less memory or at 4096 ($1000) if you have more than 8K. The first byte of screen memory holds the screen code which causes the corresponding 8 by 8 pattern to be displayed in the first column of the first row of the screen. The first 22 positions of screen memory correspond to the first row of the screen. The second 22 characters correspond to the second row and so forth.

If, for example, the screen code "1" was found in location 7680 on an unexpanded VIC, the VIC chip, set to find the 8 x 8 pattern table starting at 32768, would, in its continual scan of screen memory, find the value 1, multiply it by eight to find the displacement into the pattern table. Eight plus 32768 is 32776. It would start at that address building the dot pattern to send to the video screen. It finds the following eight bytes of data starting at 32776:

addr	dec	hex	binary
32776	24	$18	00011000
32777	36	$24	00100100
32778	66	$42	01000010
32779	126	$7E	01111110
32780	66	$42	01000010
32781	66	$42	01000010
32782	66	$42	01000010
32783	0	$00	00000000

If you look at the above pattern of ones and zeros, you will be able to see the shape of the character "A" formed by the ones on the backgraound of zeros. All of the characters' shapes are formed in the same fashion. The characters you form in your program must follow the same rules.

The information to tell the VIC chip where to find the pattern table is stored in the register at location $9005 (36869). The low-

order four bits (or low-order nybble) control the selection of the sixteen possible character set tables. The method of selecting these possible areas is as follows: Bits 3-0, the low-order nybble, of the data stored at 36869 define which area of memory is to be used for the character pattern table. The high-order nybble tells the VIC chip where screen memory is located. More about that later. The following table gives the sixteen possible values the character-table select bits may have and the memory addresses which they cause the VIC chip to use for the character patterns. The "x" in the first half of the byte means that portion of the byte does not affect the selection.

$x0	$8000	32768	Normal Upper Case/graphics
$x1	$8400	33792	Reverse of above
$x2	$8800	34816	Normal UC/LC
$x3	$8C00	35840	Reverse of above
$x4	$9000	36864	Unavailable
$x5	$9400	37888	Unavailable
$x6	$9800	38912	Unavailable
$x7	$9400	39936	Unavailable
$x8	$0000	0	Not recommended
$x9	-	-	Unavailable
$xA	-	-	Unavailable
$xB	-	-	Unavailable
$xC	$1000	4096	RAM
$xD	$1400	5120	RAM
$xE	$1800	6144	RAM
$xF	$1C00	7168	RAM

A machine language routine to accomplish the switching might be:

```
SW EQU $0C LOC 4096
SW EQU $0D LOC 5120
SW EQU $0E LOC 6144
SW EQU $0F LOC 7168
   LDA 36869 GET OLD
   AND #$F0  CLEAR LOW  - SAVE HIGH
   ORA #SW   SET LOW NYBBLE
   STA 36869 RESET
```


Of course, only one of the SW EQU's would be used, depending on which area of RAM you wanted to use for the new programmed character set. All areas which you might use for your programmed character set must be used with some thought as to the consequences. All four areas are suitable but they all fall within the memory space of BASIC programs on the unexpanded VIC and 4096 is where the screen memory is on the expanded VIC. To use them in conjunction with a BASIC program, it will be necessary to modify the BASIC pointers which identify the start and end of BASIC program memory space. These techniques are spelled out in Appendix F. 7168 is an interesting location for character memory because when it is specified, the first 128 characters are user definable but character codes 128-255 cause the VIC chip to select the standard upper case and graphics set. This can be an advantage when you wish to have both programmable characters and the standard characters. It can be a disadvantage if you are using the bit-mapped graphics techniques explained below.

To create the new characters it is necessary to build a set of 8-byte patterns in the same way the original set is constructed. It is not necessary to reserve a complete 2048 byte block of memory for a complete 256 characters if you only have a few characters you wish to ever see on the screen. However, you may not use more than one set of characters at a time. Once the VIC chip's register at 36869 ($9005) has been set, the entire screen will be generated using the character patterns found at the specified pattern table area.

Alternate Screens

Having more than one screen in memory can be a nice feature for creating special effects or saving display information for later use and other uses the fertile imagination can devise. Rapidly switching between two screens can create the effect of having a foreground and background. Whatever. The address of where the active screen is must be stored into address 36869 along with the character pattern table address. It also has a component in address 36866. The process of setting up alternate screens is less than straightforward but it still is manageable.

The starting address of screen memory has certain restrictions. It must be an even multiple of 512. This is another way of saying that the low-order byte and the low-order bit of the high-order byte of the address of the start of screen memory must all

be 0's. Furthermore, the high-order three bits of the address must be zeros. The highest address which may therefore be assigned to screen memory is 0001 1110 0000 0000 in binary, or $1E00 in hex or 7680 decimal. Of the 16 bits of the address of screen memory, only four bits may be anything other than 0. These four bits, bits 12-9, are the bits which must be stored in the VIC chip's registers to tell it where to find screen memory. Bits 12-10 get stored in the 6-4 bits of location 36869. The 9 bit gets stored in the 7-bit of location 36866. It's a little complicated but a simple set of assembler statements will make the programmer's job easier.

In the following program segment, ADR is the symbolic label for the address of the memory location where your alternate screen will be. AD1 is the value computed from ADR such that bits 12-10 of ADR end up in the proper bit positions (6-4) of the low-order byte of the generated address, with all other bits turned off (dividing an address by 1024 is the same as shifting all the bits right 10 bit positions and multiplying by 16 is the equivalent of shifting them back to the left four bit positions). AD2 is the value computed to put the 9-bit of ADR into the high-order bit of the low-order byte of AD2.

```
     ADR EQU $1000
     ADB EQU ADR/1024
     AD1 EQU ADB*16
AD2  EQU ADR/4
     LDA 36869 GET OLD VALUE
     AND #$8F  TURN OFF SEL BITS
     ORA #AD1  SET SCREEN SEL BITS
     STA 36869
     LDA 36866 GET OLD
     AND $7F   TURN OFF ONLY HIGH BIT
     ORA #AD2  SET SEL BIT
     STA 36866
```

This will work if and only if the address specified in ADR has its 9 low-order bits and its 3 high-order bits off, as required. Otherwise strange results could occur.

Some other system data fields may have to be modified if you are using the operating system of the VIC to affect the screen. The byte at location 648 tells the operating system which page the screen

resides on.

```
AD3 EQU ADR/256
    LDA #AD3
    STA 648
```

is all that is required. There are also 24 bytes of screen-wrap information (tells the operating system which lines wrap around onto the next line like in long BASIC statements) stored at locations 217 to 240 which may need to be saved and restored every time you switch screens. Saving and restoring this information is only necessary if you are using the operating system's screen display system.

Bit addressable Graphics

Bit addressable or Hi-res graphics are possible via the use of the programmable character feature. The technique used is straightforward. A block of the screen memory is pre-initialized to contain a string of sequential screen codes (the first position of screen memory contains a 0, the second a 1, the third a 2, etc.) The screen memory is not further modified by the program creating the bit addressable graphics. The character patterns (in the RAM character pattern table which is addressed by the VIC chip) are modified by the program instead of the screen memory. That is, the block of high-res screen memory always contains the same screen codes, the indexes to the pattern table. The patterns are modified a bit at a time by computing the proper bit position in the proper 8-bit row of the proper character pattern.

The size of the hi-res block is a function of available memory. The number of character patterns necessary to have a 128 bit by 128 bit hi-res screen is 256. That would be a 16 by 16 block of 8 x 8 characters. 256 characters take 2048 bytes of pattern information. To bit map the entire screen, it would take 22 x 23 or 506 distinct programmed characters. This is more than it is possible to have active at one time. There is a solution to this problem, however: double-high characters.

Double-high characters

The VIC chip has the capability of interpreting the character pattern table as a 8 x 16 bit pattern rather than 8 x 8. By setting

the low-order bit of location $9003 (36867) to 1, the VIC chip will be made to generate 8 dot wide by 16 dot high characters. To do this, sixteen bytes of character pattern information is processed for any given character code. The table of character patterns must be constructed accordingly. It is important to note that the character pattern table will now have twice as many bytes of information for the equivalent number of character codes. A full 256 character table will take 4096 bytes of RAM. This is more than is available on the unexpanded VIC. So, if you wish to do full-screen high-resolution plotting, you must have expansion memory.

The program in Appendix D sets up a 128 x 128 hi-res block in the center of the screen and is designed to allow a BASIC program to selectively turn on or off any bit in the matrix by simply poking the bit positions of the X and Y coordinates into zero page memory locations and setting a zero page indicator switch to tell the program whether to turn the bit on or off. This is very much like the sample program in the PRG except it is written in machine language and runs many many times as fast.

Color controls

There are four kinds of color controls in two categories. Border color refers to the color of the screen around the outside of the character display area of the screen. Background color refers to the color of the character display area of the screen. Foreground color refers to the color of the dots turned on within a character. Auxiliary color is applicable only for multi-color mode of character display.

The auxiliary and background colors may have all 16 possible different hues. The Border and foreground colors may have only the first eight.

Code	Color
0	Black
1	White
2	Red
3	Cyan
4	Magenta
5	Green
6	Blue
7	Yellow
8	Orange
9	Light Orange
10	Pink
11	Light Cyan
12	Light Magenta
13	Light Green
14	Light Blue
15	Light Yellow

Both border and background colors are determined by the value of the memory location $900F (36879). Background is set based on the value of the four high-order bits, bits 7-4. The border color is set in the three low-order bits, bits 2-0. The remaining bit, bit-3 in this location determines whether the normal background/foreground colors are reversed or not. If the bit is on the background remains fixed at the color specified in bits 7-4 and the foreground dots may vary from character to character under control of color memory as described next. If it is off, the background color will vary from character based on the values in character-color memory and the foreground (the dots) will all be the constant color as selected in $900F.

The foreground color is set for each character position on the screen. There is a 506 byte block of RAM reserved for the purpose of telling the VIC chip what colors to assign to the 506 screen positions. This area of RAM starts at $9600 (38400) for systems with up to 8K of RAM and at $9400 (37888) for systems with more than 8K.

Each byte of color memory has the foreground color code in the low-order three bits. This allows each character position on the screen to have its color selectable.

The high-order four bits are insignificant. The value of bit-3 is very significant, however. It selects the mode of interpretation of the character pattern. The eight byte bit pattern array for each character code can be interpreted by the VIC chip in two different ways. So far, we have only looked at the normal or "hi-res" mode of interpretation. The second mode is called multi-color mode. Bit-3 of each byte of color memory will cause the corresponding character position on the screen to be generated in hi-res mode if the bit is a 0. It will cause multi-color to be in effect for that character if it

is a 1.

Multi-color mode

In multi-color mode, the eight bytes of pattern information are not seen as eight rows of eight dots as with high-res. The eight bits of each row are seen as four two-bit color codes. This means that in multi-color mode the characters displayed on the screen consist of eight rows of four dots each. Because the characters are the same size and can be displayed at the same time as high-res characters, the four horizontal dots in each row are twice as wide as the single width dots of the hi-res characters. Another way of saying this is: the multi-color mode of character display is of half the resolution in the horizontal direction as the high-res mode.

With this mode, each character displayed may have up to four different colors. Since the mode selection bit is settable for every screen position, multi-color characters may be mixed with the single-color characters on the screen.

The two-bit color codes which define the colors and are found in the pattern table are:

00 - background
01 - border
10 - foreground
11 - auxiliary

Example:

Location $900E (36878) which controls the selection of the auxiliary for all characters contains $20. This sets auxiliary color to red.

Location $900F (36879) which controls the selection of the background color and the border color contains $4E. This sets the background color to magenta and the border color to blue.

Location 7680 ($1E00), the first character position of screen memory contains a 0 which will cause the VIC chip to find the character pattern definition at location 7168, the first pattern.

Location 38400 ($9600), the color-code table position for a screen

code of 0, contains the value $x8 (xxxx 1000) where x indicates unimportant. Signifies black foreground color and multi-color mode for the first screen character position (000 in bit positions 2-0 for black and 1 in bit position 3 for multi-color.

Locations 7168 through 7175, the character pattern definition bytes for character code 0, contain:

```
1111 1111 $FF
1010 1010 $AA
0101 0101 $55
0000 0000 $00
0001 1011 $1B
0001 1011 $1B
0001 1011 $1B
0001 1011 $1B
```

The character which will appear on the screen in the upper left corner will have a red line across the top (four double wide dots each with the auxiliary color of red). It will have black line under the red line (four codes of 10 indicate four foreground color dots). Beneath that will be a blue line (four border color dots). Beneath that will be a magenta line (four codes of 00 signify background color). Below that will be four horizontal lines of magenta, blue, black and red going from left to right.

If you understand why all that is true then you have a good grasp of color gaphics on the VIC-20.

Sticks and Sounds

Joysticks

Most input and output to and from the Vic-20 is accomplished with the assistance of two integrated circuits, the 6522 Versatile Interface Adapters (VIAs). Like the VIC chip, these two devices are in communication with the 6502 and the ROM and RAM devices via the address, data and control busses. The functioning of these chips is under control of their sixteen internal addressable registers. Like the VIC chip's registers, the VIAs'registers may be loaded and saved by programs running on the 6502. The 32 registers occupy the addresses $9110 - $912F (37136 to 37168).

The VIAs are very powerful and complex devices. A complete description of their use is beyond the scope of this text. We will present the general means of doing input and output through the VIAs and the specific means of programming for joystick and paddle use. For a complete technical description of advanced usage and programming of these devices, you should acquire the technical specifications from MOS Technology.

Each of the 6522's has two input/output ports. Each port is a set of eight electrical connections between the data bus of the VIC and world outside the VIC. The ports are called port A and port B. Each port has an associated data direction register (DDR) which is used to set the I/O port's eight lines to either input or output or a combination.

The DDR for VIA #1, port A, is found at address $9113 (37139). Each of the port's eight lines to the outside world may be configured at any time to either accept input in the form of a voltage signal or to produce output as a voltage signal. All lines represent a load of one standard TTL gate in the input mode and will drive one standard TTL load in the output mode.

The I/O lines of port A, VIA #1 are designated lines 0-7. They correspond to the bit positions 0-7 in memory location $9111 (37137). Receiving input from or putting output to any individual line is as simple as reading from or writing to that memory location. First, before actually reading or writing the data, it is necessary to tell the VIA which lines are connected to input devices and which are connected to output devices.

The DDR's eight bits correspond to the eight bits of the I/O

port. To select any given line as an input line, the corresponding bit of the DDR must be set to a 0. To identify a line of the port as an output line, the corresponding bit in the DDR must be set to a 1.

	VIA #1	VIA #2
DDR A	$9113 (37139)	$9123 (37155)
Port A	$9111 (37137)	$9121 (37153)
DDR B	$9112 (37138)	$9122 (37154)
Port B	$9110 (37136)	$9120 (37152)

Joysticks have five internal switches. They also have a source of 5 volt power which the switches direct to the appropriate pins of the connector. When the joystick is tipped in one of the four directions the switch for that direction is closed, sending the "on" signal (+5 volts or a binary 1) to the proper connector pin. If the joystick is moved to the diagonal position, both the switches are closed. The other switches send the "off" signal (0 volts or binary 0). The fifth switch is the fire button. It's signal is a 0 unless it is pressed, when it becomes a 1. The connector pins are directly wired to the VIA I/O ports. There is a "TOP" direction indicated on most joysticks. The switch associated with that direction is connected to bit-2 of VIA #1, Port A. "Bottom", then is bit-3 of VIA #1, Port A. "Left" is bit-4, same port. The fire button is bit-5, same port. "Right" is attached to bit-7 of VIA #2, Port B.

So, to sense the direction of the joystick and whether the fire button is being pushed, two I/O ports have to be input to the program. Port A of VIA #1 is used for the serial port as well as the game port and Port B of VIA #2 is used for keyboard input as well as the game port so it is advised that if you wish to use either of those devices after doing joystick input that you restore the original value of the DDRs after doing your joystick reading.

The following program segment will save the DDRs, set them for joystick input, read the joystick switches and save the switch values as bits set in storage location 0.

```
DDR1A EQU $9113
DDR2B EQU $9122
I01A  EQU $9111
I02B  EQU $9120
      LDX DDR1A
      LDY DDR2B
      LDA   #$C3   INPUT   BITS 2,3,4
      STA DDR1A
      LDA I01A
      AND #$7F  FORCE 7 BIT OFF
      STA 0
      STX DDR1A RESTORE
      LDA #$7F  INPUT BIT 7
      STA DDR2B
      LDA I02B
      AND #$80  FORCE ALL BUT 7 BIT OFF
      ORA 0     BLEND WITH REST
      STA 0
      STY DDR2B RESTORE
      RTS
```

Paddles

There may be two game paddles connected to the VIC at the same time. Each paddle has both a switch and a value which varies as the paddle is rotated. The variable value may have a value between 0 and 255. The switch will be either a one or a zero. The X paddle's variable value may be read from location $9008 (36872). The Y paddle's variable value may be read from location $9009 (36873). These are locations within the register set of the VIC chip. There is no preparation necessary to set up the reading of these values.

The switch settings of the two paddles is input in the identical manner as the switches of the joysticks. In fact, the switch for paddle X is the same bit, port and VIA as the "left" switch of the joystick. And the Y paddle switch is the same as the "right" switch. The same program as was given to read the joysticks may be used to read the paddle switches.

Sound Generation

The VIC chip has the additional function of providing sound effect capabilities. There are five registers in the VIC chip which are used to control the sound functions. Sounds are generated by the VIC chip and sent to the connected TV along with the video signal. The five registers are addressable by any program running on the VIC. They are very simple to program.

The volume control register shares location 36878 ($900E) with the auxiliary color register. The volume control is set in the low-order four bits of 36878 and the auxiliary color indicator is in the high-order four bits. The volume may be set to a value from 0 to 15, 0 setting the volume off and 15 setting it at its highest setting. The volume setting controls the volume of all four "voices"; bass, alto, soprano, and noise.

Each voice is independently controllable as to its frequency but not its volume. The high-order bit of each voice register may be considered as a voice "switch". If the bit is on (has value of 1), the voice is activated. If it is a 0 or off, the voice is deactivated. The remainder of the register, the low-order 7 bits control the frequency of the sound generated.

US TV sets, (NTSC standard), differ from European sets in the frequency range of each voice. All of the voices may be heard simultaneously if they are all switched on or any combination may be on at any one time. The general formula for equating frequencys from register values is:

$$Freq = Clock / (127 - X)$$

where X is the 7-bit register value in the range of 0 to 126. For X = 127, the formula is:

$$Freq = Clock / 128$$

The folowing table summarizes the various voices, their "Clock" values and the memory locations of their registers:

voice	address		clock NTSC (US TV's)	PAL (European)
Bass	36874	$900A	3995	4329
Alto	36875	$900B	7990	8659
Sprno	36876	$900C	15980	17320
Noise	36877	$900D	31960	34640

The theory and techniques of music synthesis and sound effects is beyond the scope of this text, but both the Programmer's Reference Guide and the VIC-20 User's Guide provide some information on creating sound effects and musical effects.

VIC Internals

The VIC-20 comes with 16K of ROM in which the internal operating programs reside. These are the programs which interpret BASIC programs and which control the input and output devices which come with the VIC and those which can be added. This block of memory extends from $C000 TO $FFFF. Some of the internal programs have been documented by Commodore in their Programmer's Reference Guide. These are what they call the "Kernal" programs. They deal mainly with input/output (I/O) processing on the VIC. This text will not attempt to duplicate the information provided in the PRG. It is once again strongly recommended that you obtain a copy of that reference work.

There are many other subroutines included in the VIC's ROM which are not covered in the PRG. We will attempt to provide information on using the more useful of those. We will also present a list of the entry points of the remainder. Those which are not discussed in detail may be decoded with the Decoder for your inspection and understanding.

Arithmetic routines

As BASIC processes your arithmetic expresions, it uses a variety of machine language subroutines to do addition, subtraction, exponentiation, trig functions, etc. These subroutines are available to the machine language program for accomplishing the same functions. They are all fairly similiar in the conventions of their use, i.e. the means of passing parameters, getting the results, etc.

Numbers in BASIC may be expressed as either integers or as "floating point" numbers. Integers have no fractional component. They are sixteen bit signed numbers which may have the range of -32767 to 32768. Negative numbers are expressed in two's complement notation as discussed in chapter four.

BASIC does all its computations in floating point mode. Floating point numbers have fractional components. They are composed of three portions, the exponent, the mantissa and the sign. The exponent occupies one byte and its binary value is 128 greater than the exponent being expressed. The value of the expressed exponent is the number of bits which the mantissa needs to be shifted. Since the exponent expression is stored in "excess 128", the range of actual exponents is -128 to 127 (stored as 0 to 255). Negative exponents

mean the mantissa (as stored) needs to be shifted to the right and positive exponents means the mantissa needs to be shifted to the left.

The mantissa is a four byte binary value. It is the shifted value of the number to be expressed. This is like scientific notation as used in physics and chemistry. The decimal system of scientific notation would express the number 12876548.765 as 1.2876548765 * 10 raised to the 7th power. In BASIC you would see this number printed as 1.2876548765 E07. A decimal exponent of 7 in scientific notation means the decimal point needs to be shifted 7 places to the right. Or that the mantissa needs to be multiplied by 10 raised to the seventh power.

It works the same way with floating point numbers except the mantissa is in binary and it needs to be multiplied by 2 raised to the power of the exponent. Multiplying a binary number by two is the same as shifting it one bit position. e.g. 6 = 0000 0110 and 12 = 0000 1100.

CBM BASIC always normalizes the mantissa before saving it in the floating point format. This means that it shifts it so the leftmost bit is always a one bit. The number 6 (binary value = 0000 0110) would be shifted so that the normalized mantissa would be 1100 0000. What goes into the exponent field is 128 plus the number of significant bit positions in the original number. 0000 0110 has three significant bit positions, so the exponent would be 131.

A few examples will be helpful. The binary representation of the floating point storage of the number 6 is:

```
1000 0011 1100 0000   0000 0000  0000 0000   0000 0000
   131        192          0           0           0
   $83        $C0        $00         $00         $00
---------  -----------------------------------------------
exponent                    mantissa
```

The exponent of 131 represents an actual exponent of 3 (131 - 128). You may consider the mantissa as a fraction with the radix point (decimal point or, rather, binary point) just to its left. The amount it must be shifted to get back to a its actual value is three bit positions. In other words, the radix point must be shifted from the left of the binary number three places to the right.

Other examples:

The number 1 (0000 0001)

```
1000 0001 1000 0000   0000 0000  0000 0000   0000 0000
   129        128          0           0           0
   $81        $80        $00         $00         $00
---------  -----------------------------------------------
exponent                    mantissa
```

The number 2 (0000 0010)

```
1000 0010 1000 0000   0000 0000  0000 0000   0000 0000
   129        128          0           0           0
   $82        $80        $00         $00         $00
---------  -----------------------------------------------
exponent                    mantissa
```

The number 3 (0000 0011)

```
1000 0010 1100 0000   0000 0000  0000 0000   0000 0000
   129        192          0           0           0
   $82        $C0        $00         $00         $00
---------  -----------------------------------------------
exponent                    mantissa
```

The number 65 (0100 0001)

```
1000 0111 1000 0010   0000 0000  0000 0000   0000 0000
   135        130          0           0           0
   $87        $82        $00         $00         $00
---------  -----------------------------------------------
exponent                    mantissa
```

The sign of the number is carried in the high-order bit of the byte following the mantissa. If the sign bit is on, the number is negative. If off it is positive.

There are two floating point accumulators maintained by BASIC, FAC1 and FAC2. FAC1 is located at $61-$66 (97-103) and FAC2 is at $69-$6E (105-111). These two accumulators are used for all mathematical operations. Following the FACs, at $6F (111), is a sign comparison flag. The high-order bit, if on, signifies the two FAC's are of differing signs.

Arithmetic routines

The following routines perform mathematical operations using the floating point accumulators FAC1 and FAC2 and values stored in other memory locations. Each routine may be executed by a JSR instruction to the indicated entry point. References to memory locations are frequently communicated to various routines by an address contained in the A-reg (LSB) and the Y-reg (MSB). We will refer to this format as format-1.

Most of the following routines use the A-reg, X-reg and Y-reg for communication. It is an interesting fact that when a SYS is done from BASIC, these three registers are loaded from memory locations 780, 781 and 782 respectively. It is therefore possible to call the following routines and those in the kernal from BASIC by SYSing to them after setting up the three register storage bytes.

Integer to FAC1 - $D391 (54161)
A two-byte integer value in format-1 is converted to a floating point number stored in FAC1.

FAC1 to Integer $D1AA (53674)
The FAC1 is converted to a two-byte integer which is saved into locations $64,$65 (100,101). The FAC1 is destroyed.

Memory to FAC1 $DBA2 (56226)
A five-byte floating point number anywhere in memory is loaded into FAC1. The address of the starting memory location is in format-1. The sign flag of FAC1 is set on if the high-order bit of the mantissa is a one, else it is set off. The exponent is returned in the A-reg.

ASCII to FAC1 $D7B5 (55221)
An ASCII string is converted to floating point format and saved in the FAC1. The string may be anywhere in memory and the address of the starting location must be pointed to by the utility string pointer at $22,$23 (34,35). The length of the string must be loaded into the A-reg.

FAC1 to ASCII $DDDD (56797)
The ASCII representation of the value in FAC1 will be saved starting at $0100 (256) and continuing until a $00 is encountered.

Memory to FAC2 $DA8C (55948)
Same as above except using FAC2 and the sign comparison flag is set. The exponent of FAC1 is returned in the A-reg.

FAC1 to Memory $DBD7 (56279)
The FAC1 is stored into any five byte memory location. The MSB of the address of the start of the memory location is passed in the X-reg. The LSB is in the Y-reg. The high-order bit of the mantissa field is forced to the FAC1 sign flag.

FAC2 to FAC1 $DBFC (56316)
A simple move is performed from FAC2 to FAC1. FAC2 is not affected.

FAC1 to FAC2 $DC0F (56335)
A simple move is performed from FAC1 to FAC2. FAC1 is not affected.

Logical AND of FAC1 and FAC2 $CFE9 (53225)
FAC1 and FAC2 are logically ANDed together, the result ending up in FAC1

Logical OR of FAC1 and FAC2 $CFE6 (53222)
FAC1 and FAC2 are logically ORed together, the result ending up in FAC1

FAC1 = FAC1 - FAC2 $D853 (55379)
FAC2 is subtracted from FAC1, the result replacing FAC1. FAC2

is not affected.

FAC1 = FAC1 + FAC2 $D86A (55402)
FAC1 is replaced by the sum of FAC1 and FAC2. It is necessary to set the sign compare flag prior to calling this routine. This is done by EORing locations $66 and $6E (102 and 110) and storing the result in $6F. It is also necessary to load the A-reg with the value found in $61 (97). Note that both of these functions are done by the Mem to FAC2 routine.

FAC1 = FAC1 * FAC2 $DA30 (55856)
FAC1 is replaced by the product of FAC1 AND FAC2. The same notes apply as for the above routine. An alternate entry point for this routine is $D828 (55848). This entry point will execute the memeory to FAC2 routine before doing the multiplication.

FAC1 = LOG (FAC1) $D9EA (55786)
FAC1 is replaced by the LOG of FAC1.

FAC1 = FAC2 / FAC1 $DB12 (56082)
FAC1 is replaced by the qoutient of FAC2 and FAC1. The same notes apply as for addition. However by JSRing to $DB0F (56079) instead, the loading of the FAC2 from memory will be accomplished prior to doing the division.

FAC1 = FAC2 ^ FAC1 $DF7B (57211)
FAC1 is replaced with FAC2 raised to the power of FAC1. Same comments as for addition. By using the $DF78 (57208) entry point, the routine to load FAC1 from memory may be executed prior to the exponentiation routine. The Memory to FAC1 routine does not properly set the sign compare flag however. Also note that when using these alternate entry points, the same setup of the A-reg and Y-reg must be performed as per Mem-to-FAC routines before calling the desired arithmetic routine.

FAC1 = FAC1 / 10 $DAFE (56062)
FAC1 is replaced by FAC1 / 10.

Compare FAC1 and Memory $DC5B (56411)
The A-reg is set depending on the result of the compare between

FAC1 and some floating point number in a specified memory location. If they are equal, the result is 0; if they are not equal the result is $FF (255). The address of the start of the memory location is in format-1.

FAC1 = ABS (FAC1) $DC58 (56408)
The FAC1 is replaced by the absolute value of FAC1.

FAC1 = INT (FAC1) $DCCC (56524)
The FAC1 is replaced by the integer portion of FAC1.

FAC1 = SGN (FAC1) $DC39 (56377)
The FAC1 is replaced by the value 0 if it was a zero, by 1 if it was greater than zero and by -1 if it was less than zero.

FAC1 = SQR (FAC1) $DF71 (57201)
The FAC1 is replaced by the square root of FAC1.

FAC1 = EXP (FAC1) $DFED (57325)
The FAC1 is replaced by the value computed by raising e of natural logarithm fame to the power of FAC1.

FAC1 = COS (FAC1) $E261 (57953)
The FAC1 is replaced by the Cosine of FAC1 expressed in radians.

FAC1 = SIN (FAC1) $DC50 (56400)
The FAC1 is replaced by the Sine of FAC1 expressed in radians.

FAC1 = TAN (FAC1) $E2B1 (58033)
The FAC1 is replaced by the tangent of FAC1 expressed in radians.

FAC1 = ATN (FAC1) $E30B (58123)
The FAC1 is replaced by the arctangent of FAC1 expressed in radians.

Input/Output routines

Most of the I/O routines are presented in the PRG but there are

two more presented here which do not appear there.

Input into BASIC buffer $C560 (50528)

The 88 byte BASIC input buffer starting at $0200 (512) is filled with characters from the keyboard. A [Return] terminates the input and a $00 signifies the end of the message in the buffer.

Output string to screen $CB1E (51998)

The starting address of a string of ASCII characters to be printed on the screen is set in format-1. The string must be terminated by a $00.

Appendix A

Mode	1	2	3	4	5	6	7	8	9	10	11	12	13
ADC		x	x	x		x	x	x			x	x	
AND		x	x	x		x	x	x			x	x	
ASL	x		x	x		x	x						
BCC										x			
BCS										x			
BEQ										x			
BIT			x			x							
BMI										x			
BNE										x			
BPL										x			
BRK									x				
BVC										x			
BVS										x			
CLC									x				
CLD									x				
CLI									x				
CLV									x				
CMP		x	x	x		x	x	x			x	x	
CPX		x	x			x							
CPY		x	x			x							
DEC			x	x		x	x						
DEX									x				
DEY									x				
EOR		x	x	x		x	x	x			x	x	
INC			x	x		x	x						
INX									x				
INY									x				
JMP						x							x
JSR						x							
LDA		x	x	x		x	x	x			x	x	
LDX		x	x		x	x		x					
LDY		x	x	x		x	x						
LSR	x		x	x		x	x						
NOP										x			

Mode	1	2	3	4	5	6	7	8	9	10	11	12	13
ORA		x	x	x		x	x	x			x	x	
PHA									x				
PHP									x				
PLA									x				
PLP									x				
ROL	x		x	x		x	x						
ROR	x		x	x		x	x						
RTI									x				
RTS									x				
SBC		x	x	x		x	x	x			x	x	
SEC									x				
SED									x				
SEI									x				
STA			x	x		x	x	x			x	x	
STX			x		x	x							
STY			x	x		x							
TAX									x				
TAY									x				
TSX									^				
TXA									x				
TXS									x				
TYA									x				

Modes:

1 - Accumulator
2 - Immediate
3 - Zero page
4 - Zero page,X
5 - Zero page,Y
6 - Absolute
7 - Absolute,X
8 - Absolute,Y
9 - Implied
10 - Relative
11 - (Indirect,X)
12 - (Indirect),Y
13 - (Indirect)

Appendix B

Dec	Hex	Binary	Dec	Hex	Binary	Dec	Hex	Binary	Dec	Hex	Binary
0	$00	0000 0000	32	$20	0010 0000	64	$40	0100 0000	96	$60	0110 0000
1	$01	0000 0001	33	$21	0010 0001	65	$41	0100 0001	97	$61	0110 0001
2	$02	0000 0010	34	$22	0010 0010	66	$42	0100 0010	98	$62	0110 0010
3	$03	0000 0011	35	$23	0010 0011	67	$43	0100 0011	99	$63	0110 0011
4	$04	0000 0100	36	$24	0010 0100	68	$44	0100 0100	100	$64	0110 0100
5	$05	0000 0101	37	$25	0010 0101	69	$45	0100 0101	101	$65	0110 0101
6	$06	0000 0110	38	$26	0010 0110	70	$46	0100 0110	102	$66	0110 0110
7	$07	0000 0111	39	$27	0010 0111	71	$47	0100 0111	103	$67	0110 0111
8	$08	0000 1000	40	$28	0010 1000	72	$48	0100 0000	104	$68	0110 1000
9	$09	0000 1001	41	$29	0010 1001	73	$49	0100 1001	105	$69	0110 1001
10	$0A	0000 1010	42	$2A	0010 1010	74	$4A	0100 1010	106	$6A	0110 1010
11	$0B	0000 1011	43	$2B	0010 1011	75	$4B	0100 1011	107	$6B	0110 1011
12	$0C	0000 1100	44	$2C	0010 1100	76	$4C	0100 1100	108	46C	0110 1100
13	$0D	0000 1101	45	$2D	0010 1101	77	$4D	0100 1101	109	$6D	0110 1101
14	$0E	0000 1110	46	$2E	0010 1110	78	$4E	0100 1110	110	$6E	0110 1110
15	$0F	0000 1111	47	$2F	0010 1111	79	$4F	0100 1111	111	46F	0110 1111
16	$10	0001 0000	48	$30	0011 0000	80	$50	0101 0000	112	70	0111 0000
17	$11	0001 0001	49	$31	0011 0001	81	$51	0101 0001	113	$71	0111 0001
18	$12	0001 0010	50	$32	0011 0010	82	452	0101 0010	114	472	0111 0010
19	$13	0001 0011	51	$33	0011 0011	83	$53	0101 0011	115	$73	0111 0011
20	$14	0001 0100	52	$34	0011 0100	84	$54	0101 0100	116	$74	0111 0100
21	$15	0001 0101	53	$35	0011 0101	85	$55	0101 0101	117	$75	0111 0101
22	$16	0001 0110	54	$36	0011 0110	86	$56	0101 0110	118	$76	0111 0110
23	$17	0001 0111	55	$37	0011 0111	87	$57	0101 0111	119	$77	0111 0111
24	$18	0001 1000	56	$38	0011 1000	88	$58	0101 1000	120	$78	0111 1000
25	$19	0001 1001	57	$39	0011 1001	89	$59	0101 1001	121	$79	0111 1001
26	$1A	0001 1010	58	$3A	0011 1010	90	$5A	0101 1010	122	$7A	0111 1010
27	$1B	0001 1011	59	$3B	0011 1011	91	$5B	0101 1011	123	$7B	0111 1011
28	$1C	0001 1100	60	$3C	0011 1100	92	$5C	0101 1100	124	$7C	0111 1100
29	$1D	0001 1101	61	$3D	0011 1101	93	$5D	0101 1101	125	$7D	0111 1101
30	$1E	0001 1110	62	$3E	0011 1110	94	$5E	0101 1110	126	$7E	0111 1110
31	$1F	0001 1111	63	$3F	0011 1111	95	$5F	0101 1111	127	$7F	0111 1111

Dec	Hex	Binary	Dec	Hex	Binary	Dec	Hex	Binary	Dec	Hex	Binary
128	$80	1000 0000	160	$A0	1010 0000	192	$C0	1100 0000	224	$E0	1110 0000
129	$81	1000 0001	161	$A1	1010 0001	193	$C1	1100 0001	225	$E1	1110 0001
130	$82	1000 0010	162	$A2	1010 0010	194	$C2	1100 0010	226	$E2	1110 0010
131	$83	1000 0011	163	$A3	1010 0011	195	$C3	1100 0011	227	$E3	1110 0011
132	$84	1000 0100	164	$A4	1010 0100	196	$C4	1100 0100	228	$E4	1110 0100
133	$85	1000 0101	165	$A5	1010 0101	197	$C5	1100 0101	229	$E5	1110 0101
134	$86	1000 0110	166	$A6	1010 0110	198	$C6	1100 0110	230	$E6	1110 0110
135	$87	1000 0111	167	$A7	1010 0111	199	$C7	1100 0111	231	$E7	1110 0111
136	$88	1000 1000	168	$A8	1010 1000	200	$C8	1100 0000	232	$E8	1110 1000
137	$89	1000 1001	169	$A9	1010 1001	201	$C9	1100 1001	233	$E9	1110 1001
138	$8A	1000 1010	170	$AA	1010 1010	202	$CA	1100 1010	234	4EA	1110 1010
139	$8B	1000 1011	171	$AB	1010 1011	203	$CB	1100 1011	235	$EB	1110 1011
140	$8C	1000 1100	172	$AC	1010 1100	204	$CC	1100 1100	236	$EC	1110 1100
141	$8D	1000 1101	173	$AD	1010 1101	025	$CD	1100 1101	237	$ED	1110 1101
142	$8E	1000 1110	174	$AE	1010 1110	206	$CE	1100 1110	238	$EE	1110 1110
143	$8F	1000 1111	175	$AF	1010 1111	207	$CF	1100 1111	239	$EF	1110 1111
144	$90	1001 0000	176	$B0	1011 0000	208	$D0	1101 0000	240	$F0	1111 0000
145	$91	1001 0001	177	$B1	1011 0001	209	$D1	1101 0001	241	$F1	1111 0001
146	$92	1001 0010	178	$B2	1011 0010	210	$D2	1101 0010	242	$F2	1111 0010
147	$93	1001 0011	179	$B3	1011 0011	211	$D3	1101 0011	243	$F2	1111 0011
148	$94	1001 0100	180	$B4	1011 0100	212	$D4	1101 0100	244	$F4	1111 0100
149	$95	1001 0101	181	$B5	1011 0101	213	$D5	1101 0101	245	$F5	1111 0101
150	$96	1001 0110	182	$B6	1011 0110	214	$D6	1111 0110	246	$F6	1111 0110
151	$97	1001 0111	183	$B7	1011 0111	215	$D7	1101 0111	247	$F7	1111 0111
152	$98	1001 1000	184	$B8	1011 1000	216	$D8	1101 1000	248	$F8	1111 1000
153	$99	1001 1001	185	$B9	1011 1001	217	$D9	1101 1001	249	$F9	1111 1001
154	$9A	1001 1010	186	$BA	1011 1010	218	$DA	1101 1010	250	$FA	1111 1010
155	$9B	1001 1011	187	$BB	1011 1011	219	$DB	1101 1011	251	$FB	1111 1011
156	$9C	1001 1100	188	$BC	1011 1100	220	$DC	1101 1100	252	$FC	1111 1100
157	$9D	1001 1101	189	$BD	1011 1101	221	$DD	1101 1101	253	$FD	1111 1101
158	$9E	1001 1110	190	$BE	1011 1110	222	$DE	1101 1110	254	$FE	1111 1110
159	$9F	1001 1111	191	$BF	1011 1111	223	$DF	1101 1111	255	$FF	1111 1111

Auto-Start Cartridges

The VIC-20 has a feature which allows a program in ROM starting at $A000 (40960) to sieze control of the machine at power-up and RESET times without any further intervention required on the part of the operator.

At power-up time one of the very first things the operating system does is check for a five character sequence starting at location $A004. If it finds $41,$30,$C3,$C2,$CD it will automatically jump to the address it found at $A000,$A001. A second address is stored in $A002,$A003. This is the address of the [RESTORE] Key processing routine.

In order to create a cartridge with Read Only Memory, it will be necessary to have a prom programmer. This is a device which "burns" PROMs. PROMs are programmable read only memorys. A PROM may be programmed or burned by applying the proper voltages and following the prescribed timing rules of the device being programmed. These functions are carefully controlled with a PROM programmer. Various sources exist for these devices and it is not horribly difficult to build your own if you have some electronic skills. Several such projects have been described in the popular computing magazines. The PRG contains the necessary information on the electrical definition of the expansion port where the cartridges attach to the VIC.

Some of the currently available cartridges actually contain BASIC programs. To do this, they do the BASIC setup routines in machine language followed by putting "RUN" followed by $13 (carriage return) in the BASIC keyboard buffer at 631-640 and jumping to the print READY routine. The following example came out of such a cartridge:

```
PLA        RESTORE  KEY PROCESSING
TAY
PLA
TAX
PLA
RTI
JSR 64909  MAINLINE PROCESSING
JSR 64850
JSR 65017
CLI       JSR 58459
```

```
        JSR 58276
        JSR 58372
        LDX #251
        TXS
        LDA #113   SET UP START OF
        STA 43      BASIC TO BE
        LDA #160      41073
        STA 44
        LDA #109
        STA 808    DISABLE STOP KEY
        LDA 40000  CHECK FOR IEEE488
        LDY #9
        CMP #162
        BEQ *+2
        LDY #3
        STY 198     # CHARS IN BUFFER
        LDX #9
LOOP    LDA TABLE,X
        STA 630,Y
        DEX
        DEY
        BNE LOOP
        JMP 50292  PRINT READY
TABLE   EQU 41056
        BYT $53D93345333A51D50D
```

The information in the table is SYS4E4:RUN[R] in abbreviated command format. The BASIC program starts at 41073. Variable storage will still be in low-memory RAM, since those pointers were never disrupted from the time of vector initialization.

VIC 20 Memory Map

Hex	Decimal	Function
0000	0-2	USR function jump
0003	3-4	Float->integer
0005	5-6	Integer->float
0007	7	Search char ":" or endline
0008	8	Scan btwn quotes flag - 00 as delimeter
0009	9	Column pos of cursor on line
000A	10	Verify flag (0=Load/1=Verify)
000B	11	Basic input buffer pointer/#subscripts
000C	12	DIM flag
000D	13	Variable flag - type:FF=string - 00=numeric
000E	14	Integer flag - type:80=integer - 00=floating pnt.
000F	15	DATA scan flag/LIST quote flag/memory flag
0010	16	Subscript flag;FNx flag
0011	17	Flags for input or read (0=input - 64=get - 152=read
0012	18	ATN sign flag:comparison evaluation flag
0013	19	Current I/O device for prompt suppress
0014	20-21	Basic integer adr.(for SYS - GOTO etc)
0016	22	Temporary string descriptor stack pointer
0017	23-24	Last temporary string vector
0019	25-33	Stack of descriptors for temporary strings
0022	34-35	Pointer for number transfer
0024	36-37	Misc number pointer
0026	38-42	Prodct area for mult
002B	43-44	Pointer to start of Basic
002D	45-46	Pointer to end of prog.start of variables
002F	47-48	Pointer to end of variables start of arrays
0031	49-50	Pointer to end of arrays
0033	51-52	Pointer to start of active strng spce(coming dwn)
0035	53-54	Pointer to top of active strings
0037	55-56	Pointer to end of memory
0039	57-58	Current Basic line number
003B	59-60	Prev BASIC line num
003D	61-62	Previous BASIC statement (for CONT)
003F	63-64	Line number - current DATA line
0041	65-66	Pointer to current DATA item
0043	67-68	Input vector

Hex	Decimal	Function
0045	69-70	Current variable name
0047	71-72	Current variable address
0049	73-74	Variable pointer for FOR/NEXT
004B	75-76	Y save/new op save/curr op pointer
004D	77	Special mask for curr oprtr;Comparison symbol
004E	78-79	Misc. work area;function def pointer hi-lo
0050	80-81	Work area;pointer to strng descrptn
0052	82	Length of above string
0053	83	Constant used by garbage collect - 3 or 7
0054	84-86	Jump vector for functions
0057	87-96	Misc. numerical storage area
0061	97-102	FAC#1
0067	103	Series evaluation constant pointer
0068	104	FAC#1 high ord propogation
0069	105-110	Accumulator #2
006F	111	Sign comparison - FAC1 vs FAC2
0070	112	Low order rounding byte for Acc#1
0071	113-114	Cassette buffer length/series pointer
0073	115-138	Subrtn:Get Basic char;7A - 7B=pointer(CHARGOT)
008B	139-143	RND storage and work area
0090	144	Status ST
0091	145	Stop Key flag: Keyswitch pia.
0092	146	Timing constant for tape
0093	147	Load or verify flag L=0/V=1
0094	148	Serial output/deferred char flag
0095	149	Serial deferred character
0096	150	Tape EOT recvd
0097	151	Register save area
0098	152	# open files
0099	153	Input device# - normally 0
009A	154	Output CMD device - normally 3
009B	155	Tape character parity
009C	156	Cassette dipole switch
009D	157	OS message flag - direct=$80 - run=0
009E	158	Cassette error pass 1
009F	159	Cassette error pass 2
00A0	160-162	Jiffy clock (HML)
00A3	163	Serial bit count

Hex	Decimal	Function
00A4	164	Cycle counterfor serial I/O
00A5	165	Cntdwn for tape write
00A6	166	Cassette buffer pointer
00A7	167	RS-232 input bit storage/Tape shrtcnt
00A8	168	RS-232 bit cnt in/ Tape read error
00A9	169	RS-232 flag start bit cK/Tape rd bit err
00AA	170	RS-232 byte buffer/Tape rd mode
00AB	171	RS-232 parity storage/Tape chKsum
00AC	172-173	Tape start addr/tape buffer / scrolling
00AE	174-175	Tape end addr/end of current program
00B0	176-177	Tape timing constants
00B2	178-179	Addr of tape buffer
00B4	180	RS-232 transmitter bit cnt out
00B5	181	RS-232 transmitter nxt bit to be sent
00B6	182	RS-232 transmitter byte buffer
00B7	183	Length of current file name string
00B8	184	Current logical file number
00B9	185	Curr secondary addr - or R/W command
00BA	186	Curr device number
00BB	187	Addr of curr file name string
00BD	189	RS-232 write shift word/Receive input char
00BE	190	#blocks remaining to read/write
00BF	191	Serial word buffer
00C0	192	Cass motor interlock
00C1	193-194	Tape start addr(load)
00C3	195-196	KERNAL setup pointer
00C5	197	Matrix co-ordinates of Key pressed
00C6	198	#of characters in Keybrd buffer
00C7	199	Reverse mode flag - 0=off - 18=on
00C8	200	End of line for input pointer
00C9	201-202	Cursor log(row - column)
00CB	203	Which Key - 64 if no Key
00CC	204	Cursor blink enabled flag - 0=on - 1=off
00CD	205	Delay before cursor blinks
00CE	206	Character under cursor
00CF	207	Cursor on/off blink flag
00D0	208	Input from screen/Keybrd
00D1	209-210	Screen addr(row)pointer(screen memory)

Hex	Decimal	Function
00D3	211	Position of cursor on curr line
00D4	212	Quote mode flag (0=off / 1=on)
00D5	213	Line length for screen (22/44/66/88)
00D6	214	Current screen line number
00D7	215	ASCII value of last key press
00D8	216	# of inserts outstanding
00D9	217-241	Screen line link table
00F2	242	Screen row marker
00F3	243-244	Screen color ptr
00F5	245-246	Keyscan table indirect
00F7	247-248	Pointer to RS-232 receive buffer addr
00F9	249-250	Pointer to RS-232 transmitter buffer addr
00FB	251-254	Free zero page locations
00FF	255	BASIC storage
0100	256-266	Floating to ASCII work area
0100	256-318	Tape error log
0100	256-511	Processor stack area
0200	512-600	Basic input buffer
0259	601-610	Logical file number table
0263	611-620	Device number table
026D	621-630	Secondary addr of R/W cmd - table
0277	631-640	Keyboard buffer
0281	641-642	Start of memory
0283	643-644	Top of memory
0285	645	Serial timeout flag
0286	646	Active color code
0287	647	Original color under cursor
0288	648	Screen page
0289	649	Keyboard buffer max length
028A	650	Repeat flag - 0=cursor only - 128=all keys
028B	651	Delay before repeat occurs
028C	652	Delay btwn repeats
028D	653	Shift flag byte
028E	654	Last shift pattern
028F	655-656	Indirect for keyboard table setup
0291	657	Shift mode switch - 0=enabled - 128=locked
0292	658	Auto scroll dwn flag(0=on - <>0=off)
0293	659	RS232 control register

Hex	Decimal	Function
0294	660	RS232 Command register
0295	661-662	Non standard(bittime/2-100)
0297	663	RS-232 status register
0298	664	Number of bits to send
0299	665-666	Baud rate full bit time
029B	667	RS-232 end of receiver pointer
029C	668	RS-232 start receive buffer
029D	669	RS-232 start transmit output buf
029E	670	RS-232 end of transmit buffer
029F	671-672	Holds IRQ during tape operation
02A1	673-767	Program indirects
0300	768-769	Indirect error routine
0302	770-771	Indirect warm start
0304	772-773	Indirect crunch BASIC
0306	774-775	Indirect token print
0308	776-777	Indirect new token
030A	778-779	Indirect symbol evaluation
030C	780	Temporary storage during SYS of A-reg
030D	781	Temporary storage during SYS of X-reg
030E	782	Temporary storage during SYS of Y-reg
030F	783	Temporary storage during SYS of P-reg
0314	788-789	IRQ vector
0316	790-791	BRK vector
0318	792-793	NMI vector
031A	794-795	Open logical file vector
031C	796-797	Close logical file vector
031E	798-799	Set input device vector
0320	800-801	Set output device vector
0322	802-803	Reset default I/O
0324	804-805	Input from device
0326	806-807	Output to device vector
0328	808-809	Test STOP key vector
032A	810-811	Get from keyboard vector
032C	812-813	Close all files vector
032E	814-815	Basic USR command vector
0330	816-817	Load from device vector
0332	818-819	Save to device vector
033C	828-1019	Cassette buffer

Hex	Decimal	Function
0400	1024-4095	3K expansion RAM area
1000	4096-7679	User Basic area
1E00	7680-8191	Screen memory
2000	8192-16383	8K expansion RAM/ROM block 1
4000	16384-24575	8K expansion RAM/ROM block 2
6000	24576-32767	8K expansion RAM/ROM block 3
1000	4096-4607	Screen memory (RAM > 8K)
1200	4608-	User Basic area (RAM > 8K)
9400	37888-38399	Colour RAM (RAM > 8K)
8000	32768-36863	4K character generator ROM
8000	32768-33791	Upper case and graphics
8400	33792-33815	Reversed upper case and graphics
8C00	35840-36863	Reversed upper and lower case
9000	36864-37887	I/O BLOCK 0
9000	36864-36879	Address of VIC chip registers
9000	36864	bits 0-6:horiz centrng/bit 7:interlace
9001	36865	vertical centering
9002	36866	bits 0-6:# cols/bit 7 part of scrn mem
9003	36867	bits 1-6:# rows/bit 0 sets 8x8 or 16x8 chars
9004	36868	TV raster beam line
9005	36869	bits 0-3 start of character memory
9006	36870	Horizontal position of light pen
9007	36871	Vertical position of light pen
9008	36872	Digitized value of paddle X
9009	36873	Digitized value of paddle Y
900A	36874	Frequency for oscillator 1 (low)
900B	36875	Frequency for oscillator 2 (medium)
900C	36876	Frequency for oscillator 3 (high)
900D	36877	Frequency of noise source
900E	36878	bits 0-3:volume/4-7:auxilary color
900F	36879	Screen and border color register
9110	37136-37151	6522 VIA No.1
9110	37136	Port B output register(user port RS232)
9111	37137	Port A output register
9112	37138	Data direction register B
9113	37139	Data direction register A
9114	37140	Timer 1 low byte
9115	37141	Timer 1 high byte

Hex	Decimal	Function
9116	37142	Timer 1 low byte
9117	37143	Timer 1 high byte
9118	37144	Timer 2 low byte
9119	37145	Timer 2 high byte
911A	37146	Shift register
911B	37147	Auxiliary control register
911C	37148	Peripheral control register
911D	37149	Interrupt flag register
911E	37150	Interrupt enable register
911F	37151	Port A (sense cassette switch)
9120	37152-37167	6522 VIA No. 2
9120	37152	Port B output register
9121	37153	Port A output register keyboard row scan
9122	37154	Data direction register B
9123	37155	Data direction register A
9124	37156	Timer 1. low byte latch
9125	37157	Timer 1. high byte latch
9126	37158	Timer 1. low byte counter
9127	37200	Timer 1. high byte counter
9128	37160	Timer 2.low byte latch
9129	37161	Timer 2.high byte latch
912A	37162	Shift register
912B	37163	Auxiliary control register
912C	37164	Peripheral control register
912D	37165	Interrupt flag register
912D	37165	Interrupt enable register
912F	37167	Port A output register
9400	37888-38399	Location of COLOR RAM (RAM > 8K)
9600	38400-38911	Normal location of COLOR RAM
9800	38912-39935	I/O block 2
9C00	39936-40959	I/O block 3
A000	40960-49151	Expansion ROM
C000	49152-57343	BASIC
C000	49152	Keyword action addresses
C046	49222	Function action addresses
C074	49268	Operator action addresses
C092	49298	Keyword Table
C193	49555	Error messages

Hex	Decimal	Function
C38A	50058	FOR - GOSUB search stack
C3B8	50104	Open memory space
C3FB	50171	Test stack depth
C408	50184	Check available memory
C435	50229	Send error message
C474	50292	Print READY.
C483	50307	New BASIC line processing
C533	50483	BASIC line chaining
C560	50528	Receive line from keyboard
C57C	50556	Tokenize BASIC line
C613	50707	Search for line number
C642	50754	Perform NEW
C660	50784	Perform CLR
C68E	50830	Reset BASIC execution to start-of-program
C69C	50844	Perform LIST
C742	51010	Perform FOR
C7ED	51181	Execute BASIC statement
C81D	51229	Perform RESTORE
C82C	51244	Perform STOP and END
C857	51287	Perform CONT
C871	51313	Perform RUN
C883	51331	Perform GOSUB
C8A0	51360	Perform GOTO
C8D2	51410	Perform RETURN
C8EB	51435	Perform DATA
C906	51462	Scan for next statement
C909	51465	Scan for next line
C928	51496	Perform IF
C93B	51515	Perform REM
C94B	51531	Perform ON
C96B	51563	Get integer from text
C9A5	51621	Perform LET
CA80	51840	Perform PRINT#
CA86	51846	Perform CMD
CA9A	51866	Perform PRINT
CB1E	51998	Print string from any memory
CB3B	52027	Print format character
CB4D	52045	Process bad input

Hex	Decimal	Function
CB7B	52091	Perform GET
CBA5	52133	Perform INPUT#
CBBF	52159	Perform INPUT
CBF9	52217	Prompt & input
CC06	52230	Perform READ
CCFC	52476	Input error messages
CD1E	52510	Perform NEXT
CD78	52600	Type match check
CD9E	52638	Evaluate expression
CEA8	52904	PI in floating point
CEF1	52977	Evaluate within parenthesis
CEF7	52983	Check for ")"
CEFA	52986	Check for "("
CEFD	52989	Check for " - "
CF08	53000	Syntax error
CF14	53012	Search for variable name
CFA7	53159	Set up FN references
CFE6	53222	Perform OR
CFE9	53225	Perform AND
D016	53270	Comparison routine
D07E	53374	Perform DIM
D08B	53387	Locate variable
D113	53523	Check for alpha ASCII
D11D	53533	Create new variable
D194	53652	Array pointer routine
D1A5	53669	32768 in floating point
D1BF	53695	FAC1 to integer
D1D1	53713	Find or Create Array
D34C	54092	Compute subscript size
D37D	54141	Perform FRE
D391	54161	Integer to FAC1
D39E	54174	Perform POS
D3A6	54182	Check for DIRECT mode
D3B3	54195	Perform DEF
D3E1	54241	Check FN syntax
D3F4	54260	Evaluate FN
D465	54373	Perform STR$
D475	54389	Calculate string vector

Hex	Decimal	Function
D487	54407	Set up string
D4F4	54516	Build string vector
D526	54566	Collect garbage (make room for string)
D5BD	54717	Check string collection eligibility
D606	54790	Collect string
D63D	54845	Concatenate string
D67A	54906	Build string to memory
D6A3	54947	Discard unwanted string
D6DB	55003	Clean the descriptor stack
D6EC	55020	Perform CHR$
D700	55040	Perform LEFT$
D72C	55084	Perform RIGHT$
D737	55095	Perform MID$
D761	55137	Pull string parameters from stack
D77C	55164	Perform LEN
D782	55170	Wxit string mode
D78B	55179	Perform ASC
D79B	55195	Input byte parameter
D7AD	55213	Perform VAL
D7EB	55275	Get POKE/WAIT parameters
D7F7	55287	FAC1 to integer
D80D	55309	Perform PEEK
D824	55332	Perform POKE
D82D	55341	Perform WAIT
D849	55369	Add 0.5 to FAC1
D850	55376	Perform subtraction
D86A	55402	Perform addition
D947	55623	Complement FAC1
D97E	55678	Overflow
D983	55683	Single byte multiply
D9BC	55740	Floating point constants
D9EA	55786	Perform LOG
DA28	55848	Mulitply FAC1 * memory
DA30	55856	Multiply FAC2 * FAC1
DA59	55897	Multiply a bit
DA8C	55948	Memory to FAC2
DAB7	55991	Adjust FAC1/FAC2
DAD4	56020	Underflow/overflow

Hex	Decimal	Function
DAE2	56034	Multiply FAC1 by 10
DAF9	56057	Constant 10
DAFE	56062	Divide by 10
DB07	56071	Divide FAC2 / memory
DB0F	56079	Divide memory / FAC1
DB12	56082	Divide FAC2 / FAC1
DBA2	56226	Memory to FAC1
DBD7	56247	FAC1 to memory
DBFC	56316	FAC2 to FAC1
DC0F	56335	FAC1 to FAC2
DC1B	56347	Round off FAC1
DC2B	56363	Get sign
DC39	56377	Perform SGN
DC58	56408	Perform ABS
DC5B	56411	Compare FAC1 to memory
DC9B	56475	FAC1 to integer
DCCC	56524	Perform INT
DCF3	56563	ASCII to FAC1
DD7E	56702	Get new ASCII digit
DDB3	56755	Constants
DDDD	56797	FAC1 to ASCII
DF11	57105	More constants
DF71	57201	Perform SQR
DF78	57208	Perform exponentiation
DFB4	57268	Perform negation
DFBF	57279	More constants yet
DFED	57325	Perform EXP
E040	57408	Series evaluation
E08A	57482	RND constants
E094	57492	Perform RND
E261	57953	Perform COS
E268	57960	Perform SIN
E2B1	58033	Perform TAN
E2DD	58077	Constants for trig functions
E30B	58123	Perform ATN
E33B	58171	Constants for ATN
E378	58232	Initialize RAM vectors
E387	58247	CHRGET for zero page

Hex	Decimal	Function
E3A4	58276	Initialize BASIC
E429	58409	Messages
E44F	58447	Vector initialization
E467	58487	Warm restart
E476	58502	Program patch area
E4A0	58528	Serial output "1"
E4A9	58537	Serial output "0"
E4B2	58546	Get serial input & clock
E4BC	58556	Program patch area
E500	58624	Set 6522 addrs
E505	58629	Set screen limits
E50A	58634	Track cursor location
E518	58648	Initialize I/O
E54C	58700	Normalize screen
E55F	58719	Clear screen
E581	58753	Home cursor
E587	58759	Set screen pointers
E5BB	58811	Set I/o defaults
E5C3	58819	Set vic chip defaults
E5CF	58831	Input from keyboard
E64F	58959	Input from screen
E6B8	59064	Quote mark test
E6C5	59077	Set up screen print
E6EA	59114	Advance cursor
E715	59157	Retreat cursor
E72D	59181	Back into previous line
E742	59202	Output to screen
E8C3	59587	Go to next line
E8D8	59608	Do 'RETURN'
E8E8	59624	Check line decrement
E8FA	59674	Check line increment
E912	59666	Set colour code
E921	59681	Colour code table
E929	59689	Code conversion
E975	59765	Scroll screen
E9EE	59886	Open space on screen
EA56	59990	Move screen line
EA6E	60014	Synch colour transfer

Hex	Decimal	Function
EA7E	60030	Set start-of-line
EA8D	60045	Clear screen line
EAA1	60065	Print to screen
EAAA	60074	Store on screen
EAB2	60082	Synch colour to char
EABF	60095	Interrupt (IRQ)
EB1E	60190	Check keyboard
EC00	60416	Set text mode
EC46	60486	Keyboard vectors
EC5E	60510	Keyboard maps
ED21	60705	Graphics/text control
ED30	60720	Set graphics mode
ED5B	60763	Wrap up screen line
ED6A	60778	Shifted key matrix
EDA3	60835	Control key matrix
EDE4	60900	Vic chip defaults
EDFD	60925	Screen line adds low
EE14	60948	Send 'talk'
EE17	60951	Send 'listen'
EE1C	60956	Send control char
EE49	61001	Send to serial bus
EEB7	61111	Timeout on serial
EEC0	61120	Send listen SA
EEC5	61125	Clear ATN
EECE	61134	Send talk SA
EEE4	61156	Send serial deferred
EEF6	61174	Send 'untalk'
EF04	61188	Send 'unlisten'
EF19	61209	Receive from serial bus
EF84	61316	Clock line on
EF8D	61325	Clock line off
EF96	61334	Delay 1 ms
EFA3	61347	RS232 send (NMI)
EFEE	61422	New RS232 byte send
F016	61462	Error or quit
F027	61479	Compute bit count
F036	61494	RS232 receive (NMI)
F05B	61531	Setup to receive

Hex	Decimal	Function
F09D	61597	Receive parity error
F0A2	61602	Receive overrun error
F0A5	61605	Receive break error
F0A8	61608	Receive frame error
F0B9	61625	Bad device
F0BC	61628	File to RS232
F0ED	61677	Send to RS232 buffer
F116	61718	Input from RS232 buffer
F14F	61775	Get from RS232 buffer
F160	61792	Check serial bus idle
F174	61812	Messages
F1E2	61922	Print if direct
F1F5	61941	Get..
F205	61957	..from RS232
F20E	61966	Input
F250	62032	Get..tape/serial/RS232
F27A	62074	Output..
F290	62096	..to tape
F2C7	62151	Set input device
F309	62217	Set output device
F34A	62282	Close
F3CF	62415	Find file
F3DF	62431	Set file values
F3EF	62447	Abort all files
F3F3	62451	Restore default I/O
F40A	62474	Do file opening
F495	62613	Send SA
F4C7	62663	Open RS232
F542	62786	Load program
F647	63047	'SEARCHING'
F659	63065	Print file name
F66A	63082	'LOADING/VERIFYING'
F675	63093	Save program
F728	63272	'SAVING'
F734	63284	Bump clock
F760	63328	Get time
F767	63335	Set time
F770	63344	Action stop key

Hex	Decimal	Function
F77E	63358	File Error Messages
F7AF	63407	Find any tape header
F7E7	63463	Write tape header
F84D	63565	Get buffer address
F854	63572	Set buffer start - end pointers
F867	63591	Find specific header
F88A	63626	Bump tape pointer
F894	63636	'PRESS PLAY'
F8AB	63659	Check cassette status
F8B7	63671	'PRESS RECORD'
F8C0	63680	Initiate tape read
F8E3	63715	Initiate tape write
F8F4	63732	Common tape read/write
F94B	63819	Check tape stop
F95D	63837	Set timing
F98E	63886	Read bits (IRQ)
FAAD	64173	Store characters
FBD2	64466	Reset pointer
FBDB	64475	New tape character setup
FBEA	64490	Toggle tape
FC06	64518	Data write
FC0B	64523	Tape write (IRQ)
FC95	64661	Leader write (IRQ)
FCCF	64719	Restore vectors
FCF6	64758	Set vector
FD08	64776	Kill motor
FD11	64785	Check read/write pointer
FD1B	64795	Bump read/write pointer
FD22	64802	Powerup entry
FD3F	64831	Check A-rom
FD52	64850	Set Kernal2
FD8D	64909	Initialize system constants
FDF1	65009	IRQ vectors
FDF9	65017	Initialize I/O regs
FE49	65097	Save data name
FE50	65104	Save file details
FE57	65111	Get status
FE66	65126	Flag ST

Hex	Decimal	Function
FE6F	65135	Set timeout
FE73	65139	Read/set top memory
FE82	65154	Read/set bottom of memory
FE91	65169	Test memory location
FEA9	65193	NMI interrupt entry
FED2	65234	RESET/STOP warm start
FEDE	65246	NMI RS232 sequences
FF56	65366	Restore & exit
FF5C	65372	RS232 timing table
FF72	65394	Main IRQ entry
FF8A	65418	Jumbo jump table
FFFA	65530	Hardware vectors

Sample Bit-Mapped plotting

The following programs are an example of a machine language subroutine, callable by either BASIC or machine language, and a BASIC program which incorporates and uses the routine. The machine language program is designed to turn on individual pixels based on a x and y bit-position passed from the calling routine. The routine assumes that the character pattern table has been set up to start at location 5120. This is set as an EQU labeled CH. Changing the EQU is all that is necessary to change the routine to use another area. It also assumes that the x and y bit positions have been stored as numbers in the range of 0-127 in locations 1 and 2. The upper left corner of the screen is considered pit position 0,0 and the lower right is position 127,127. Location zero is used as a mode switch. If the value contained there is greater than 127, the pixel at the specified coordinates will be set on. If it is less than 128 the mode is erase and the dot will be turned off.

The BASIC program which calls the plot routine has the responsibility of setting up the screen parameters. The character memory is set up to begin at 5120 and the screen memory is set up to begin at 4096. The screen size is set at 16 lines by 16 columns. All of these functions are accomplished with the four POKEs in line 105. The foreground color is set at black for all scren positions by statement 110. The screen codes are pre-initialized to 0-255, the top left corner of the screen having value 0 and the bottom right having 255. This is done in statement 130. Statement 140 and 150 are the mainline of the program and simply cause a line to be drawn from the upper left corner to the lower right and be erased repetitively.

The BASIC program:

```
     105 POKE 36869,205: POKE36864,11: POKE 36865,36: POKE 36866,16:
POKE 36867,32
     110 FOR I = 37888 TO 37888 + 505: POKE I,0: NEXT
     120 FOR I = 5120 TO 7167: POKE I,0: NEXT
     130 FOR I = 4096 TO 4096 + 255: POKE I, I - 4096: NEXT
     140 POKE 0,128: GOSUB 150: POKE 0,0: GOSUB 150: GOTO 140
     150 FOR I = 0 TO 127: POKE 1,I: POKE 2,I: SYS 9000: NEXT:
RETURN
```

The machine language program:

```
MODE EQU 0
VX   EQU 1
VY   EQU 2
LO   EQU 3
HI   EQU 4
CH   EQU 5120
     EQU 9000
     LDA #0
     STA ←HI
     LDA ←VY
     AND #$F8          8 * INT(Y/8)
     LDX #3      *
LP   ASL A       *
     ROL ←HI     * ROW-DISP = 16*(8*INT(Y/8))
     DEX         *
     BPL LP      *
     STA ←LO
     LDA ←VX
     AND $F8           COL-DISP = 8 * INT (X/8)
     ADC ←LO     *
     STA ←LO     *      ROW + COL
     BCC H       *
     INC ←HI     *
H    LDA ←VY
     CLC
     AND #7      *     CHAR PATTERN ROW
     ADC ←LO
     BCC H2
     INC ←HI
H2   CLC
     ADC #CH      *
     STA ←LO      *
     LDA ←HI      * ADDRESS = CH+DISPLACEMENT
     ADC #CH/256 *
     STA ←HI      *
     LDA ←VX
     AND #7             BIT POSITION
     TAX
```



```
          LDY #0
          TYA
          SEC          *
     ROR  ROR A        *  POSITION THE BIT
          DEX          *
          BPL ROR      *
          BIT ←MODE       SET OR RESET
          BMI ORA
          EOR #$FF     *
          AND (LO),Y   *     RESET
          STA (LO),Y   *
     ORA  ORA (LO),Y
          STA (LO),Y
          RTS          *  RETURN TO BASIC
```

Explanation:

The screen has 16 rows of 16 characters each. Each character has eight rows of eight dots each. The screen memory contains a sequential set of character codes, from 0 to 255. The first sixteen of the screen codes index the character patterns which will appear on the top line, the second sixteen referencing the patterns which will appear on the second line, etc. So, character pattern #0 is the pattern for the upper left eight by eight block of dots and character pattern #1 is the block to the right of that and character pattern #16 is the pattern of dots for the block below it and so forth till character pattern #255 which describes which pixels will be illuminated in the bottom right corner of the screen.

The above machine language subroutine computes which pattern corresponds to any given X and Y dot coordinate. It also computes which byte within the eight-byte pattern holds the dot in question and which bit position within that byte to turn on or off.

The Y value represents how far from the top of the screen the dot is located. If it is in the range of 0-7, the dot is in the first row of characters. If is is 8-15, it is in the second row, etc. So to find the row Y must be divided by 8.

Each row of characters contains 16 patterns across the screen and each pattern contains eight bytes of data. So, for every row, the position within the table increases by 8 times 16 or 128.

The address of the appropriate byte of the table to modify is built in the two-byte field labeled LO and HI. Statements 10-17 compute the displacement into the table due to the Y value. Statement 11 strips the low-order three bits from the Y-value. This accomplishes both the initial division by 8 to find the row-number and the multiplication by eight. If we now multiply this "stripped" value of Y by 16, we will get the "row-displacement" into the character table. This is exactly what statements 12-16 do. This is a four-bit left shift. Every time a number is shifted left one bit the number increases in value by a factor of two. So, a four-bit shift is a multiply-by-16 operation. Note that the carry bit is the communication between the two bytes of the shift operation. The bits which come off the left of the A-reg get shifted into the byte at HI.

The X value tells us how far into the row we must go to get the proper character within the row. An X value of 0-7 would address the first character within the row. A value of 8-15 would address the second, etc. Dividing the X-value by 8 gives the character position within the row. Each character has eight bytes of information associated with it in the pattern table, so we have to multiply the character position by eight to get the displacement into the row. Statement 19 effectively accomplishes both functions at the same time. It strips the low-order three bits, which is like shifting right three bits (division by eight) then shifting back left again (multiplying by eight). This character position within the row is added to the "row-displacement" previously computed based on the Y-value. The result goes back into HI-LO.

Next, the byte within the character pattern must be computed and added into the displacement value being built. Each byte within the character pattern description describes a different row of bits or pixels on the screen. The Y value contains the row information. In fact, the low-order three bits may be seen to be the "character row" value. They may have the value of 0-7. To strip off the high-order bits, the instruction at statement 26 is employed. This value is then added to the displacement.

Now, in statements 30-35, the address of the start of the character pattern table is added to the displacement within the table to get the address of the actual byte to modify.

The only thing remaining is to get the actual bit within the byte and either turn it on or off as indicated by the mode switch in location 0. The low-order three bits of the X-value may be considered

the bit-position. Statements 36-38 strip off the high-order bits and transfer the value into the X-reg where it will be used as a counter. Statements 41-44 shift the carry bit into the A-reg. The carry bit will end up the same number of bit positions from the left as the value created in the X-reg. This byte can now be used to modify the byte we have just computed the position of. The BIT instruction tests the high-order bit of location 0 to see if a set or a reset of the specified bit is required. If it is a reset, the bits in the A-reg are flipped so the AND instruction can set the proper bit off. Recall that AND needs zeros in the bit positions which need to be turned off and ones in the bit positions which need to remain unchanged.

If a "set" is requested, the ORA instruction will accomplish the deed. Finally, we return from whence we were called by the RTS.

Alternate Load Addresses

BASIC Programs may be readily loaded into any piece of RAM large enough to accomodate them. In fact, many different BASIC programs may be in memory at the same time but in different places. The program which is "current" in the system at any one time is the one which is pointed to by the "start-of-BASIC" pointer at 43,44. Loading a program into an alternate location is simply a matter of changing 43,44 to point to the address where you wish the program to load.

To get the program to RUN in the alternate space requires an additional operation. The byte immediately preceeding the program load address must have the value "$00". Without doing this, RUNning the program will produce a SYNTAX? ERROR.

So, the procedure is:

```
POKE 44,xxx/256: POKE 43,xxx-PEEK(44)*256: POKE xxx-1,0:CLR
```

where xxx is the address you wish to load the program into.

When SAVEing a program, the entire memory space between the addresses at 43,44 and 45,46 will be SAVEd. It is thus possible to SAVE multiple programs as one large memory load.

UPGRADE OFFER

There are three upgrades possible to the standard version of Develop-20.

If you have a cassette-based version, you may exchange it for the diskette version. The diskette-based version supports Assembly and load module files on diskette as opposed to the cassette-based version which will only support cassette files.

The extended-feature version requires at least 8K to run (minimum expansion memory of 3K). Its main benefits are the ability to switch rapidly between Editor and Assembler modes without having to save the Assembly program then load the Assembler which then will have to re-load the Assembly program. The Editor and Assembler are combined into one program. The other feature provided with the extended-feature version is the ability to directly POKE the machine language output from the Assembler into the VIC's memory without having to save it as a Load module for input by the Loader. The POKE option is added to and does not replace the SAVE feature of the standard version.

The cost of the extended-feature upgrade is $10.00. RS-232 Printer suport is available with the extended-feature version for an additional $5.00. The disk version upgrade is $5.00. All feature option prices are additive. So disk and extended features and RS-232 is $20.00.

To receive the upgrades, you need only return the original cassette or diskette along with your check or money order (U.S. Funds) or VISA/MC number/expiration date.

Send the magnetic media and the payment to:

French Silk
P.O. BOX 207
Cannon Falls, MN 55009

If you do not wish to take advantage of the exchange offer but would like to be on the mailing list for future products, send us your name and address. We will be happy to include you in our future mailings.